SHARP

SHARP
COMPUTER
SOFTWARE

X68000用

C COMPILER PRO-68K ver2.0

アセンブラマニュアル

X68000用

C COMPILER PRO-68K ver2.0

アセンブラマニュアル

SHARP

はじめに

本書は、X68000オペレーティングシステムHuman68k　ver. 2.0のもとで動作する「XAssembler」、「XLinker」、「XDebugger」、「XArchiver」、「XLibrarian」、「XConverter」の取り扱い説明、及びアセンブラを用いたソフトウェア開発についてまとめています。

なお、別冊の「Cユーザーズマニュアル」にXCコンパイラの商品構成、、動作環境の作成方法、各マニュアルの内容が説明されています。初めて本プログラムをご使用になる方は、必ず「Cユーザーズマニュアル」を先にご覧ください。

Cコンパイラ（XC)、BASIC-Cコンバータ（XBAStoC）によるプログラムの開発においてもコンパイルの途中で一度アセンブラを通ることになります。これにより、アセンブラを含めた複合的なソフトウェア開発が可能となっています。また、アセンブリ言語だけのプログラムによっても、効率のよい実行プログラムを作成できます。

本書の構成

このマニュアルは、全体で２部に分かれています。
第１部は「ユーザーズガイド」です。
第２部はアセンブリ言語の「リファレンスマニュアル」です。

第１部　ユーザーズガイド

アセンブリ言語を中心とした、機械語レベルのソフトウェア開発を行うときに必要となるソフトウェア（アセンブラ、リンカ、デバッガ、アーカイバ、ライブラリアン、コンバータ）の扱いかたについて説明しています。

第２章と第３章は入門者向けなので、すでに機械語レベルのソフトウェア開発に詳しいかたは、お読みになる必要はありません。

第１章　はじめに

使用前に知っておくことをまとめています。
はじめにお読みください。

第２章　開発の過程

機械語レベルのソフトウェア開発とはどのようなことか、また、その手順について基本的なことを説明します。

第３章　プログラム開発入門

アセンブリ言語を使ったソフトウェアの開発がどのようなものかを知っていただくために、簡単な例を示しました。
開発に必要なソフトウェアをどのように使用するのかという例としてお読みください。

第４章　アセンブラ

アセンブラ（XAssembler）の取扱い説明です。
アセンブリ言語についてのリファレンスは、第２部にまとめました。

第5章　リンカ

アセンブラによって生成されたオブジェクトコードから、実行可能プログラムを作成するのがリンカです。
この章では、リンカ (XLinker) の使用方法を説明しています。

第6章　デバッガ

デバッガは、プログラムを検証・テストするツールです。
この章では、デバッガ (XDebugger) の使用方法について説明しています。

第7章　アーカイバ

プログラムの開発の過程で、複数個の文書ファイルが作成される場合があります。
このような複数個のファイルを、ファイルの種類ごとにひとまとめに整理して、管理を行うツールがアーカイバです。
ソースプログラムやドキュメントファイルなどの文書ファイルを、まとめて管理するのが主な用途になります。
この章では、アーカイバ (XArchiver) の使用方法を説明しています。

第8章　ライブラリアン

複数個のファイルを、種類ごとにまとめて管理するツールがライブラリアンです。
アーカイバと機能がよく似ていますが、リンクする複数のファイルを管理するのが主な用途といえます。
この章では、ライブラリアン (XLibrarian) の使用方法を説明しています。

第9章　コンバータ

リンカによって作成された実行可能ファイルの形式を変更するツールがコンバータです。
この章では、コンバータ (XConverter) の使用方法を説明しています。

第2部　リファレンスマニュアル

第1章　アセンブラ

XAssemblerのリファレンスマニュアルです。
アセンブリ言語についての規則を詳しくまとめています。

第2章　命令セット一覧

MPU68000の命令セットを一覧できるようまとめています。
アセンブリ言語によるプログラム作成のときご活用ください。

CONTENTS

第1部　ユーザーズガイド

第1章　はじめに

第2章　開発の過程

第3章　プログラム開発入門

第4章　アセンブラ

第5章 リンカ

第6章 デバッガ

第7章 アーカイバ

第8章 ライブラリアン

第9章　コンバータ

第2部　リファレンスマニュアル

第1章　アセンブラ

第2章　命令セットリファレンス一覧

索　　引

なお、本書においては、命令などの書式（フォーマット）の説明にあたり、以下の記号を用いることとします。

<　>　指定が必要であることを示します。
[　]　省略が可能であることを示します。
・・・　任意の回数だけ繰り返すことを示します。

第 1 部

ユーザーズガイド

第1章

はじめに

ソフトウェア開発キットの目的
バックアップの作成
コマンドモードによる使用

本アセンブラマニュアルでは、C compiler PRO-68K のことをソフトウェア開発キットと呼んでいます。

X68000 は、メイン CPU に MPU68000 を使用しています。
MPU68000 は、ミニコンピュータやワークステーションに多く採用されてきました。
この幅広い実績が示すように、MPU68000 は、16 ビットマイクロプロセッサの中でも汎用性があり高度な機能を備えているものの一つです。
また MPU68000 は、大型コンピュータが採用している 32 ビット CPU に代表される強力なアドレッシングモードや、体系的なプログラミングを可能にするシステム制御命令、また考え得るあらゆる事態に対応できる例外処理などを実現しています。
X68000 は、この優れた CPU を中心に、安全性に優れ高速アクセスを実現したフロッピーディスク装置や操作を簡易にするマウス、先進の ADPCM（音声のデジタル録音）などの各種デバイスが装備さています。
ソフトウェア開発キットは、X68000 のこれらの高度なハードウェア機能を、最大限に引き出すソフトウェア開発のために用意されたものです。

1.1 ソフトウェア開発キットの目的

ソフトウェア開発キットによって、次のことが可能となります。

独立して実行可能なプログラム作成

X68000に搭載されているMPU68000は、機械語という2進数の羅例からなる言語のみを理解し実行することができます。
ここでいう「独立して実行可能なプログラム」とは、この機械語のプログラムのことを指します。
さて、この機械語はマイクロプロセッサにとっては唯一、直接実行できる言語ですが、人間にとってはほとんど理解できない数列であり、この言語でプログラムを組むことはほとんど不可能といえます。
そこで、機械語の代わりにアセンブリ言語という言語を使用することができます。
アセンブリ言語は機械語の命令と1対1で対応した命令セットをもつ言語で、以下にその例を示します。

アセンブリ言語		機械語
move. l	(a1), d5	0010101000010001
add. w	d0, d1	1101001001000000
trap	#15	0100111001001111

上記のようにアセンブリ言語は、人間にとって2進数の羅列より分かりやすい文字列で構成されています。
しかし直接マイクロプロセッサに実行させることはできません。
そこでアセンブリ言語で記述されたプログラムを、機械語に変換する必要があります。
そのためのツールがアセンブラ（AS. X）です。なお、X68000ではこのアセンブラとリンカ（LK. X）というツールにより、機械語への変換とOS上で実行可能なプログラムの作成を行います。

さてアセンブリ言語は、機械語の命令と1対1で対応しているという点で、人間にとって理解しにくいという欠点があります。

そこで、通常は高級言語と呼ばれる、より人間にとって分かりやすい、そして開発効率の高い言語を使用することになります。
X68000では、BASICとCが高級言語として用意されています。

BASICで記述したプログラムは、通常BASICインタープリタ（BASIC.X）と呼ばれる実行可能プログラムを用いて実行します。
前述したアセンブラと違い、BASICインタープリタはBASICで記述したプログラムを機械語に変換するのではなく、実行時にその記述内容を解析し、BASICインタープリタ内のサブルーチンやOSが用意しているファンクションコールなどを呼び出します。
つまり、BASICで記述したプログラムを実行するとき、マイクロプロセッサはそのプログラム自身を実行しているのではなく、すでに機械語に変換されているBASICインタープリタなどの実行可能プログラムを実行しているのです。
したがって、BASICで記述されたプログラムを実行するには常にBASICインタープリタが必要であり、独立してBASICのプログラムを実行することはできません。
このため、他のプログラムと組み合わせて実行したい時などで不便なことがあります。
また、実行時にプログラムを解析するため、実行速度は期待できる速度にはなりえません。

ソフトウェア開発キットの目的として、BASICやCで記述されたプログラムを「独立して実行可能なプログラム」に変換する、ということがあります。
C言語で記述されたプログラムを例にとって説明します。
C言語で記述されたプログラムは、Cコンパイラ（CC. X）と呼ばれる実行可能プログラムにより、機械語に変換することができます。
ただし、この機械語のモジュールはまだ実行可能プログラムではありません。
実行できるようにするには、リンクという処理が必要です。
リンクとは、Cコンパイラやアセンブラにより生成された1つまたは複数の機械語のモジュール（オブジェクト）を結合して、1つの実行可能プログラムを生成することです。
リンカ（LK. X）はこのような処理を行い、さらに実行可能プログラムを実行可能ファイルとしてディスクなどに格納します。
この実行可能ファイルは、OSであるHuman68kの管理下で、正常にディ

スクからメモリへロード・実行できるように、プログラム本体に実行するための情報が付加されています。
こうして、作成された実行可能プログラムを実行するには、もはやCコンパイラやアセンブラ、リンカは必要ありません。
実行速度も、マイクロプロセッサが直接実行できる機械語に変換済みなので、高速です。
なお、BASICで記述されたプログラムはC言語へ自動変換することで、実行可能プログラムに変換することができます。

まとめてみると、C言語で記述したプログラムを実行するまでには、コンパイル➡リンクいう2つの段階を経る必要があります。
このように、2つの段階に分ける利点としては、以下のようなことが挙げられます。

●プログラムを複数に分割し、リンクの時に再びまとめて実行可能プログラムを作成することができる。

プログラムを複数に分割すると、プログラム全体の見通しが良くなることと、MAKEコマンド（プログラマーズマニュアル参照）により、バグの見つかったソースプログラムのみを修正・再コンパイルした後、全てのプログラムを再リンクするだけで済むので、開発時間が短縮できます。

●1つの実行可能プログラムを複数の言語で記述したプログラムより作成することができる。

Cコンパイラが作成するアセンブリ言語は、人間が最初からアセンブリ言語で記述するより多少冗長になります。
当然、実行速度もCコンパイラの出力するアセンブリ言語のほうが遅くなります。
したがって、プログラム中で高速性を重視する部分をC言語で記述すると、十分な速度が得られなくなる場合があるかも知れません。
そんなときには、速度が要求される部分だけ直接アセンブリ言語で記述し、その他の部分をC言語で記述することで問題は解決します。
アセンブリ言語のプログラムの方はアセンブルのみ、C言語のプログラムの方はコンパイル・アセンブルを行い、リンクすれば1つの実行可能プログラムが作成されます（ただし、アセンブリ言語のプログラムは、C言語の関数として呼び出せるように設計されなければなりません）。

ソフトウェア開発キットには、よりプログラムの作成とデバッグを効率よく

行うために、デバッガやライブラリアン、アーカイバ、そしてコンバータなどのツールが用意されています。

1.2 バックアップの作成

バックアップをしましょう

フロッピーディスクの中身は少しずつですが摩擦で消耗します。
また、コーヒーをフロッピーディスクにこぼしてしまった、などということもないとはいえません。
このような場合、データは失われることがあります。
そこで大切なデータをこのような事故や誤った操作から保護するためバックアップをする必要があります。
大切なディスクは作業の前に必ずバックアップをとる習慣をつけてください。
通常の作業ではこのバックアップディスクを使います。
もとのディスクは万一に備えて大切に保管しておきます。
ディスクのバックアップは以下の手順で行います。
この時、手順を誤ると、バックアップするデータを消失する場合もあるので、十分注意してください。

●コピー先のフロッピーディスクをフォーマットします。

①まず、まだフォーマットされていないフロッピーディスクか、またはフォーマットしても構わない（データが消失します）フロッピーディスクをBドライブにセットします。
②カレントドライブを、Bドライブ以外（ここではAドライブ）に設定し、formatコマンドを実行します。
formatコマンドの実行方法は以下の通りです。

```
A＞format B:
```

もし、formatコマンドが実行できなくてエラーが発生した場合は、PATHコマンドで、formatコマンドのあるディレクトリへのパスをコマンド検索パスに加えてください。

③フォーマットを開始します。
format コマンドを起動してからの操作方法は、「Human68k ユーザーズマニュアル」を参照してください。
④フォーマットが正常に終了したら、次にディスクコピーを行います。

●ディスクコピー

⑤次のように diskcopy コマンドを実行します。
この例では A ドライブから B ドライブへコピーします。

```
A>diskcopy A: B:
```

⑥ diskcopy が正常に実行できると、画面に起動メッセージが表示され、キー入力待ちになります。
そこで、キーを押す前に、A ドライブにバックアップしたいディスクをセットします。
B ドライブはフォーマットしたディスクをセットします。
⑦画面に表示されている指示にしたがって、ディスクコピーを実行します。
diskcopy の操作方法は、「Human68k ユーザーズマニュアル」を参照してください。
⑧正常にディスクコピーが終了したら、B ドライブのディスクに A ドライブのディスクの内容がコピーされているはずです。
今後はこのコピーしたディスクを使用します。
A ドライブのディスクはマスターディスクとして大切に保管してください。

1.3 コマンドモードによる使用

ソフトウェア開発キットの各ソフトウェアは通常コマンドモードで使用します。

「コマンドモード」とは、キーボードから入力される文字列を「コマンド」と解釈し、そのコマンドを実行するモードです。
コマンドモードではCOMMAND. Xという名前のプログラムが、キーボードからの入力を受けて、各コマンドをHuman68k上で実行します。
ユーザーは用意されているコマンドを使用することによって、ソフトウェア開発を行うことができます。

コマンドモードへの切り替え

コマンドモードへの切り替え方法には次の2種類があります。

- ビジュアルシェルでCOMMAND. Xを選択する。
- 起動時にコマンドモードに入るようにする。

1.　ビジュアルシェルから入る

ビジュアルシェルからHuman68kのコマンドモードに入るには、デスクトップ画面から"COMMAND. X"を実行します。
"COMMAND. X"はHuman68kのコマンド処理プログラムで、実行するとコンピュータをコマンドモードに設定します。
ポインタを"COMMAND. X"のアイコンに重ね、そこでマウストラックボールの左ボタンをダブルクリックしてください。
ビジュアルシェルが消え、画面はHuman68kのコマンドモードになります。

画面に

```
A>■
```

が現れます。

1.3 コマンドモードによる使用

画面に表示された"A＞"は、「プロンプト」と呼ばれる記号です。
プロンプトとは、英語で「うながす」という意味の言葉で、まさにコンピュータがユーザーに対してなんらかの命令（コマンド）の入力を促していることを示しています。
コマンドモードでは、このプロンプトに続けて目的のコマンドをキーボードから入力しながら、作業を進めていきます。

コマンドモードから再びビジュアルシェルに戻るには、「EXIT コマンド」を実行します。
画面にプロンプト("A＞")表示されていることを確かめて、次のように入力してください。
入力は、半角なら大文字でも小文字でもかまいません。

A＞EXIT⏎（⏎は、リターンキーを押す動作を表す）

リターンキーを押してしばらくすると画面にメッセージが表示されますので、それに従ってください。
画面には再びビジュアルシェルが表示されます。

2. 起動時にコマンドモードに入る

起動時にコマンドシェルに入るかビジュアルシェルに入るかは、システムディスク中にある"CONFIG. SYS"というファイルに記述されている"TITLE＝"の指定により決定します。
X68000は起動されると、まず「システムファイル」と呼ばれるHuman68kを構成するプログラムをメモリに読み込んで実行します。
Human68kが起動すると、"CONFIG. SYS"というファイルを読み込み、その内容を解析します。そして"TITLE＝"の後に記述されているファイルの中でビジュアルシェルを起動するように設定してあれば、ビジュアルシェルが起動します。
"TITLE＝"の後に記述されているファイルがディスクに存在しなかったり、存在してもコマンドシェルを起動するようになっていたり、また、そのファイルが文書ファイルの形式であった場合にコマンドシェルが起動します。
つまり、

- 起動したシステムディスクのルートディレクトリにある"CONFIG. SYS"に記述されている"TITLE＝"で指定されたファイルが存在していて、そ

の中でビジュアルシェルを起動するように設定してあれば、ビジュアルシェルが起動されます。

●"TITLE="で指定されたファイルが存在しなければ、コマンドシェルが起動されます。

ただし、"CONFIG. SYS"の中で"TITLE="の指定がない場合は"TITLE=¥TITLE. SYS" と記述されているものとみなされ、上記の判断が行われます。
ビジュアルシェルが起動するディスクでは"CONFIG. SYS"の中で"TITLE=¥TITLE. SYS" のように指定されています。
そこで、コマンドシェルを起動するには、"TITLE. SYS"が入っているディレクトリのウィンドウを開き、コマンドメニューの"Rename"を使って、"TITLE. SYS" というファイル名を"TITLE. VS"という名前に変えてください。
あるいはコマンドシェルの状態で次のように入力してください。

```
A>REN  ¥CONFIG¥TITLE. SYS  TITLE. VS
```

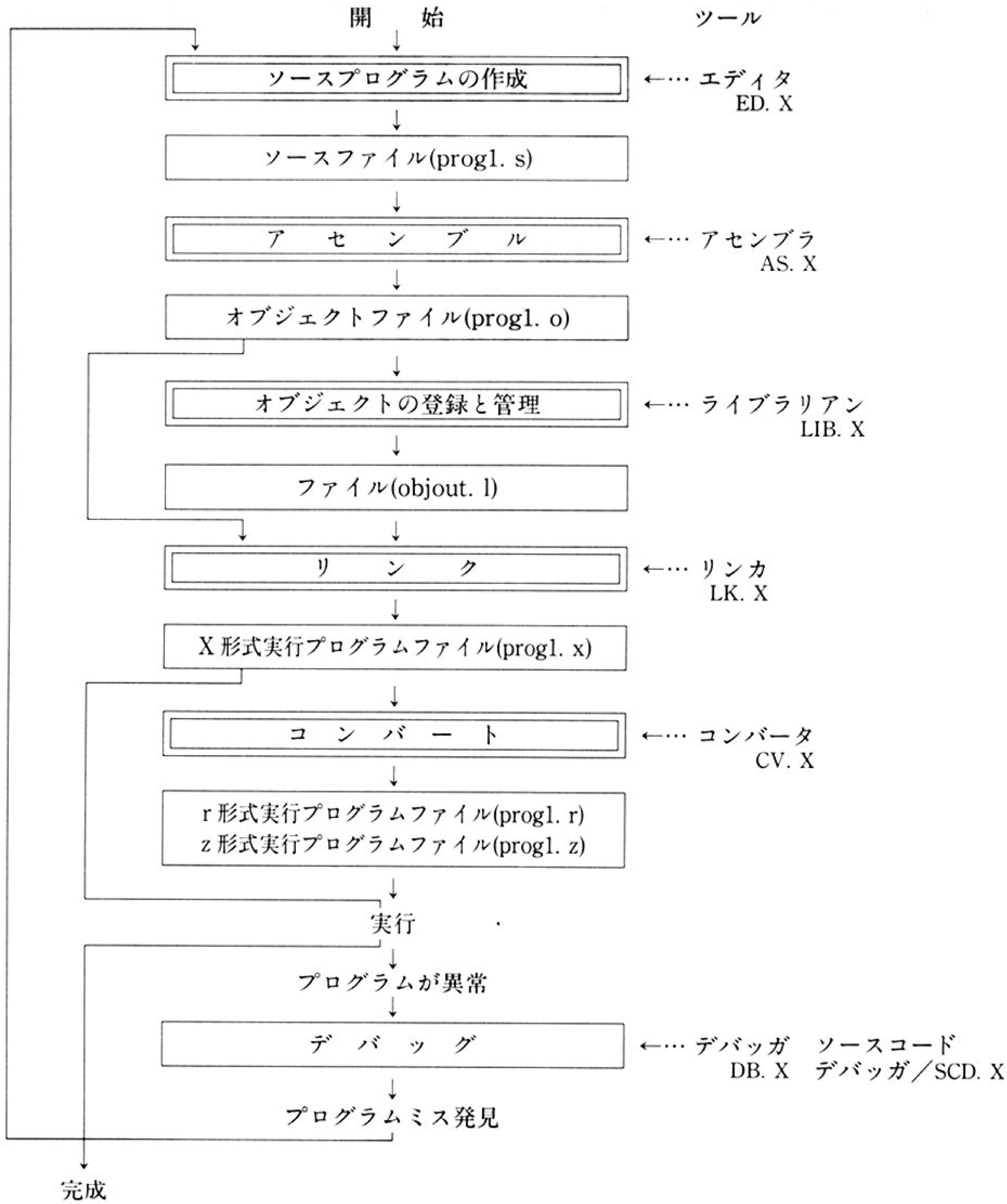
再びビジュアルシェルを起動するには、この逆（TITLE. VS から TITLE. SYS への Rename）を行います。
また、"CONFIG. SYS"の中の"TITLE="の記述を変更すれば、コマンドシェルを起動することができます。つまり、存在しないファイルを"TITLE=" で指定すれば Human68k はタイトルが存在しないと判断し、コマンドシェルを起動します。
"CONFIG. SYS"の内容を変更するにはエディタ"ED. X"を使用します。
"ED. X"の使い方は Human68k Ver. 2.0 ユーザーズマニュアルの"ED. X"の解説の項を参照してください。

第 2 章
開発の過程

開発の過程

高級言語を使用する場合

本章では、プログラム開発が実際にどのようなものか、手順に沿って解説します。
また（　）内にファイル名の例 prog1 で示します。



2.1 開発の過程

ソフトウェアの開発は、実行したい作業（処理）の流れを分析し、これをプログラムに置き換えることです。
もちろん分析だけでは、実行が可能なプログラムはできあがりません。
分析した内容を次にアルゴリズムとしてソースプログラムに表現し、それから X68000 が実行できる型にするということが必要になります。

これが「プログラム開発」ということです。

プログラムの開発ではこのようにソースプログラムをコーディングする時点から、どのような開発環境があり、どのようにして実行可能なプログラムを作成するのかを、知っておく必要があります。
これを簡単に説明します。

Human68k オペレーティングシステムで実行可能プログラムを作成するまでの過程には、一定の手順があります。
これは大きく分けて、次の6つのステップから成り立っています。

1. **ソースプログラムの作成**　(ED. X)
2. **アセンブル**　(AS. X)
3. **オブジェクトの登録と管理**　(LIB. X)
4. **リンク**　(LK. X)
5. **コンバート**　(CV. X)
6. **デバッグ**　(DB. X/SCD. X)

各ステップには、その手順に適したツールが用意されています。
カッコ内は、対応するツールの名称です。

2.1.1 ソースプログラムの作成

はじめに、ソースプログラムをディスク上にソースファイルとして作成します。
ソースプログラムは、コンピュータ言語を使ってアルゴリズム（処理の内

容）を表現したものです。
コンピュータ言語はいろいろな記号を使っていますが、人間も読むことができます。
これを可読性といいます。
ソースプログラムは可読性のあるプログラムです。
Human68k オペレーティングシステムの TYPE コマンドで表示できます。

ソフトウェア開発キットでは、ED というエディタ (ED. X) を用いてこのソースプログラムの作成作業を行います。

ED はフルスクリーンエディタなので、カーソルをスクリーン上で自由に動かすことができ、効率のよい編集が可能です。
ED を次のように起動します。
詳しい説明は「Human68k ユーザーズマニュアル」を参照してください。

ED［＜スイッチ＞］＜ファイル名＞

ファイル名は、編集するソースプログラムファイルの名称を入力します。
パス名で指定されるディレクトリ中に、そのファイルがあれば、それを編集します。
なければ＜ファイル名＞で指定される名称のファイルを新規に作成します。
拡張子のところは、3 文字までに任意に指定が可能ですが、アセンブラのソースプログラムの場合は小文字の s にしておきます。

2.1.2 アセンブル

ソースプログラムは、アセンブリ言語と呼ばれる言語で作成します。
これは CPU が直接解釈できる機械語と一対一に対応しています。
機械語は人間には無意味な数値の集合にしか見えません。
アセンブリ言語は、機械語の意味内容を人間にわかるように簡単な単語に置き換えたものです。しかし、アセンブリ言語のままでは、CPU にとって無意味なものになってしまいます。
そこでアセンブラというプログラムを使って、これを機械語に直します。

この作業をアセンブルといい、機械語に翻訳したものを、オブジェクトプログラムと呼びます。

2.1　開発の過程

アセンブラは次のように起動します。

AS [<スイッチ>] <ソースファイル名>

ここで単に AS と入力するとヘルプメッセージが表示されます。
ここでヘルプメッセージを見てください（スイッチの説明参照)。

```
X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
使用法：as [スイッチ] ファイル名
        /t path         テンポラリーパス指定
        /o name         オブジェクトファイル名
        /i path         インクルードパス指定
        /p [file]       リストファイル作成
        /n              最適化の禁止
        /w              ワーニングエラーの出力禁止
        /u              未定義シンボルを外部参照にする
        /d              すべてのシンボルを外部定義にする
        /8              シンボルの識別長を8バイトにする
        /m nn           最大シンボル数(270<nn<32768)
        /s symbol       シンボルの定義
        /x [file]       シンボルの出力
        /a              絶対ショートアドレス形式対応モード
```

ソースプログラムのファイル名を入力します。
ファイル名の入力にあたって、拡張子が指定されなかった場合には、拡張子として小文字の s が使用されます。
これをデフォルト（省略時解釈）といいます。
たとえば、sample. s がソースプログラムのファイル名だったとき、ここで

　　sample

と、

　　sample. s

とは同じ意味になります。

アセンブルを行うとオブジェクトが生成されます。
オブジェクトのファイル名は、拡張子を小文字の o としてソースファイル名と同じものになります。

sample. s　ソース
↓
アセンブル
↓
sample. o　オブジェクト

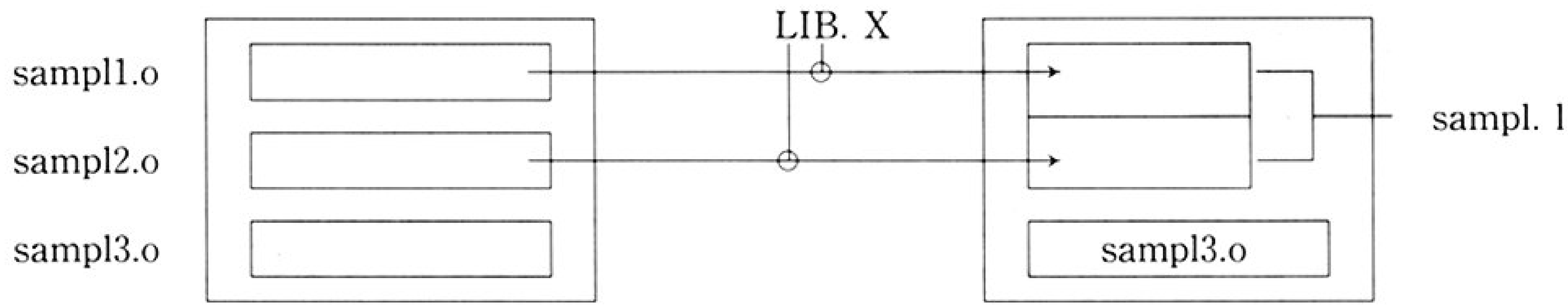
アセンブルのときに/pというスイッチを指定するとリストファイルが作られます。
リストファイルはアセンブルのさまざまな情報を含んでいます。

2.1.3 オブジェクトの登録・管理

プログラミングには、多くの労力と時間を必要とします。
一度開発したプログラムのうち、別のプログラムにも使えるものがあるならば、それを活用するのが合理的です。
このため、多くの場合、まとまりのある処理ごとに分割してプログラミングする手法が用いられています。
この分割した部分を「モジュール」といいます。

リンカにはこの分割したモジュールの結合機能があります。

モジュールに分割することで、ファイルの数は増えます。
そこで複数のプログラムの開発を、かぎられたディスクを使って同時進行するような場合には、ファイルの整理がつかなくなりがちです。
ライブラリアン（LIB.X）は、このような分割化の長所と一本化の長所を、ともに活かすためのツールです。
分割アセンブル、あるいは分割コンパイルしたオブジェクトファイルを拡張子lで示されるライブラリファイルに登録します。
ライブラリファイルには、複数のファイルを登録できますから、ディレクトリ上からは、あたかも1つのファイルとして管理することができます。



また、同様にソースファイルの登録・管理については、アーカイバAR. Xを使用します。

2.1.4 リンク

アセンブルされたプログラムには、未解決シンボル（実際のメモリ位置が指定されていないシンボル）などが残っていますので、そのままでは実行できません。
また、分割コンパイルされたオブジェクトモジュールは、1本のプログラムにまとめる必要があります。

この実行可能にする変換と結合の作業を「リンク」といいます。
リンカによってリンクを行います。

リンカは、リロケータブル、すなわち主記憶上のどの領域に配置されても実行可能な、1つ以上のオブジェクトファイルを結合して、1本の実行可能プログラムを生成します。
また、ライブラリファイルに登録されたライブラリルーチンを検索して結合します。
リンカによって生成された実行可能プログラムは、拡張子を小文字のxとしたファイルになります。
つまりリンカの処理によって、実行可能型ファイルとして、リロケータブルなx形式の実行可能プログラムが生成されます。
x形式のプログラムにはリロケーション情報が含まれているので、実行するときHuman68kオペレーティングシステムが自動的に再配置を行います。
また、x形式のプログラムではプログラム自体も大きくなっています。

2.1.5 コンバート

x形式のプログラムは、先頭につけられたリロケーション情報を削除し、フルリロケータブルなr形式の実行可能プログラムに変換することができます。
この作業にはコンバータ（CV. X）を用います。
Human68kの実行型ファイルにはx形式、z形式、r形式の3つのタイプがあります。

コンバータはx形式を変換してr形式またはz形式を作成します。

x形式では、コード、スタック、データなどが、メモリ上で、4つの独立した領域に割あてられます。
したがって、それぞれを参照するさい、混同がなく、再配置が可能な長所があります。
一般に高級言語をコンパイルした場合には、管理性のよさから、こちらのメモリの形式が自動的に選択されます。
z形式は、絶対アドレスにロードされ、この固定番地にロードされないと実行することはできません。
r形式は、フルリロケータブル形式ともいい、メモリのどの位置にローディングしても実行が可能です。
このうち、どちらを選択するかによって、ソースプログラム上でのメモリの表現の仕方が違ってきますので注意してください。

注：プログラムの設計上すべてのx形式ファイルがr形式に変換できるわけではありません。

2.1.6 デバッグ

リンク、コンバートが終わり、実行可能なプログラムが作成されました。
しかし、いざ実行してみても、意図どおりの結果が得られない場合があります。
今までの各過程において、文法エラーなどは厳しくチェックされてきました。
しかし、アセンブラやリンカがチェックするのは、CPUがプログラムを解

釈・実行できるかどうか、ということにかぎられています。
演算の順番を間違えたり、無限ループを組んでしまった場合には、CPU が実行できるため、エラー検出の対象にはならないのです。

このときに、デバッガを用いてプログラムのミスを発見して、ソースプログラムを修正します。

2.2 高級言語を使用する場合

ソフトウェア開発キットのソフトウェアはアセンブリ言語による開発の他に、次の高級言語と併せて使用します。

C 言語　　(XC)

アセンブラによる開発の過程は、C 言語による開発でも必要となります。
C 言語で書いたソースプログラムは C コンパイラというプログラムによってアセンブラソースに変換されます。
いったんアセンブラソースに変換された後は、これまでに述べた開発の流れに組み込まれます。

BASIC 言語　　(XBAStoC)

BASIC 言語ソースを C 言語ソースに変換するコンバータです。
X-BASIC インタプリタ用のソースプログラムを C 言語のソースに変換できます。
そこで BASIC 言語で開発を行うときは、

```
BASIC 言語ソース
     ↓   ←コンバータ (XBAStoC)
 C 言語ソース
     ↓   ←コンパイラ (XC)
アセンブラソース
     ↓
┌─────────────┐
│ アセンブラによる │
│  開発の過程   │
└─────────────┘
```

という流れで作業を行います。

第3章

プログラム開発入門

プログラム仕様の決定
ソースファイルの作成
アセンブラの使用
リンカの作業
実　　行
設計の修正と変更
デバッガ
ま と め

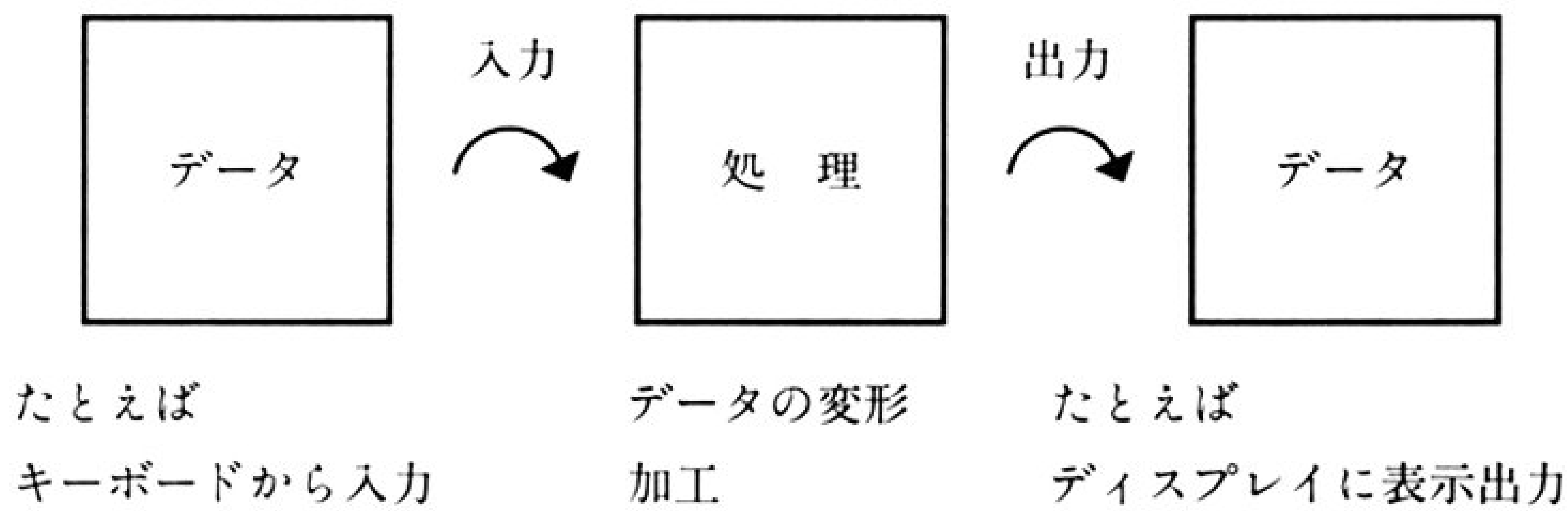
この章では、「ソフトウェア開発キット」を使用して簡単なプログラムを作成する手順をプログラム開発の例として示します。

なお、アセンブリ言語による開発の一般的な手順をご存じのかたは、この章を読む必要はありません。
また、開発の過程を説明するうえで、ためしに、いくつかの基本的なエラーやバグを出してみます。
エラーやバグはないほうがいいのですが、開発の過程ではよく発生します。
この章は完全なプログラムの作成例というのではなく、プログラム開発手順の例としてお読みください。
それでは、アセンブリ言語でプログラムを作ってみましょう。

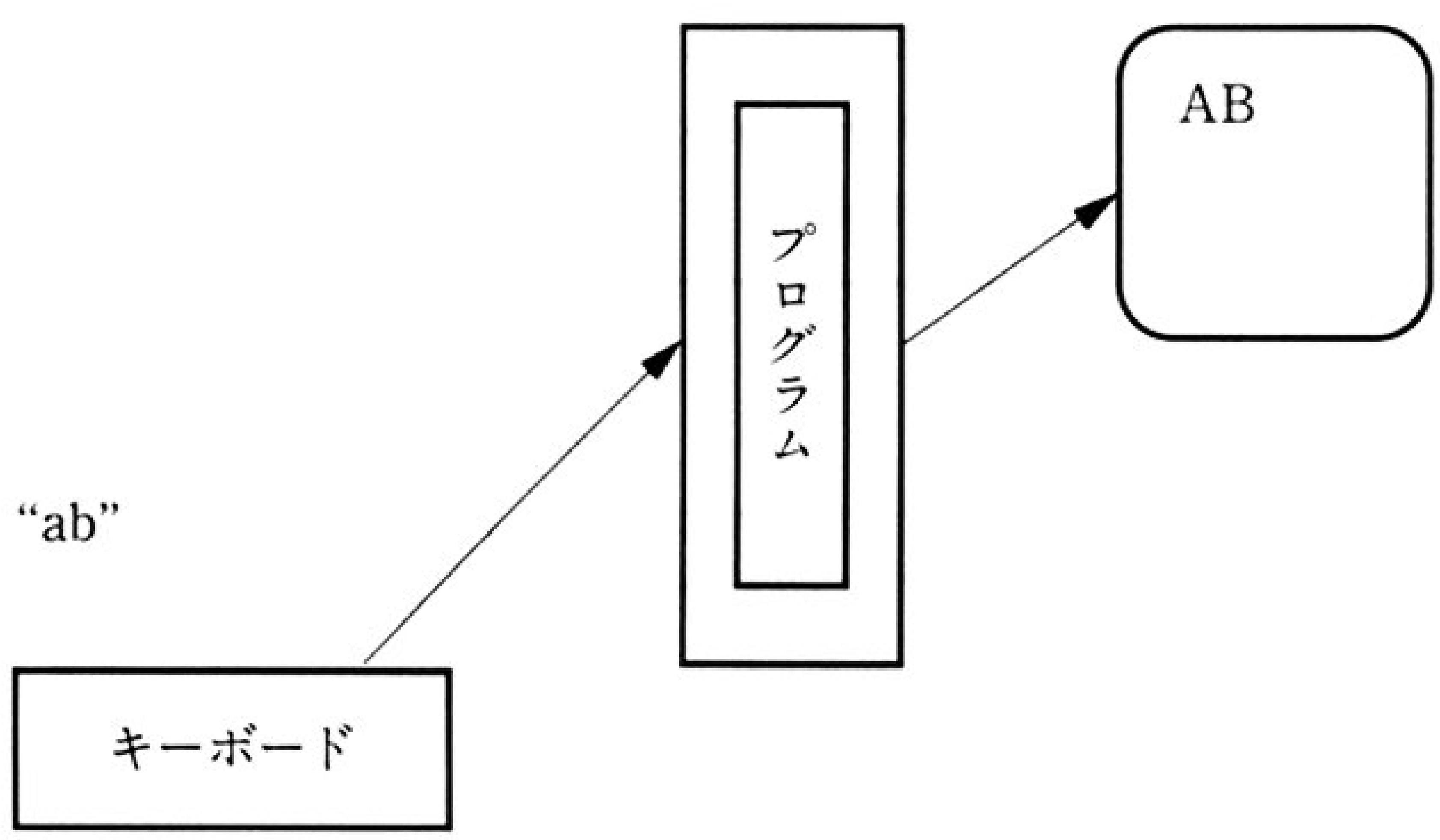
3.1 プログラム仕様の決定

3.1.1 設　計

プログラムには目的があります。
まずどのような「処理」をさせるプログラムなのかを決めなくてはなりません。
これが「プログラムの設計」です。
または、「仕様を決める」ともいいます。
ここで、「処理」を簡単にいうと、入力したデータを変形して、別のデータとして出力する過程のことです。



ここでは、小文字の英数字を入力すると、大文字に変換されて出力されるプログラムというものを考えてみます。
CAPSキーの働きをするプログラムといえます。
仮に、このプログラムの名前をcapとしておきましょう。

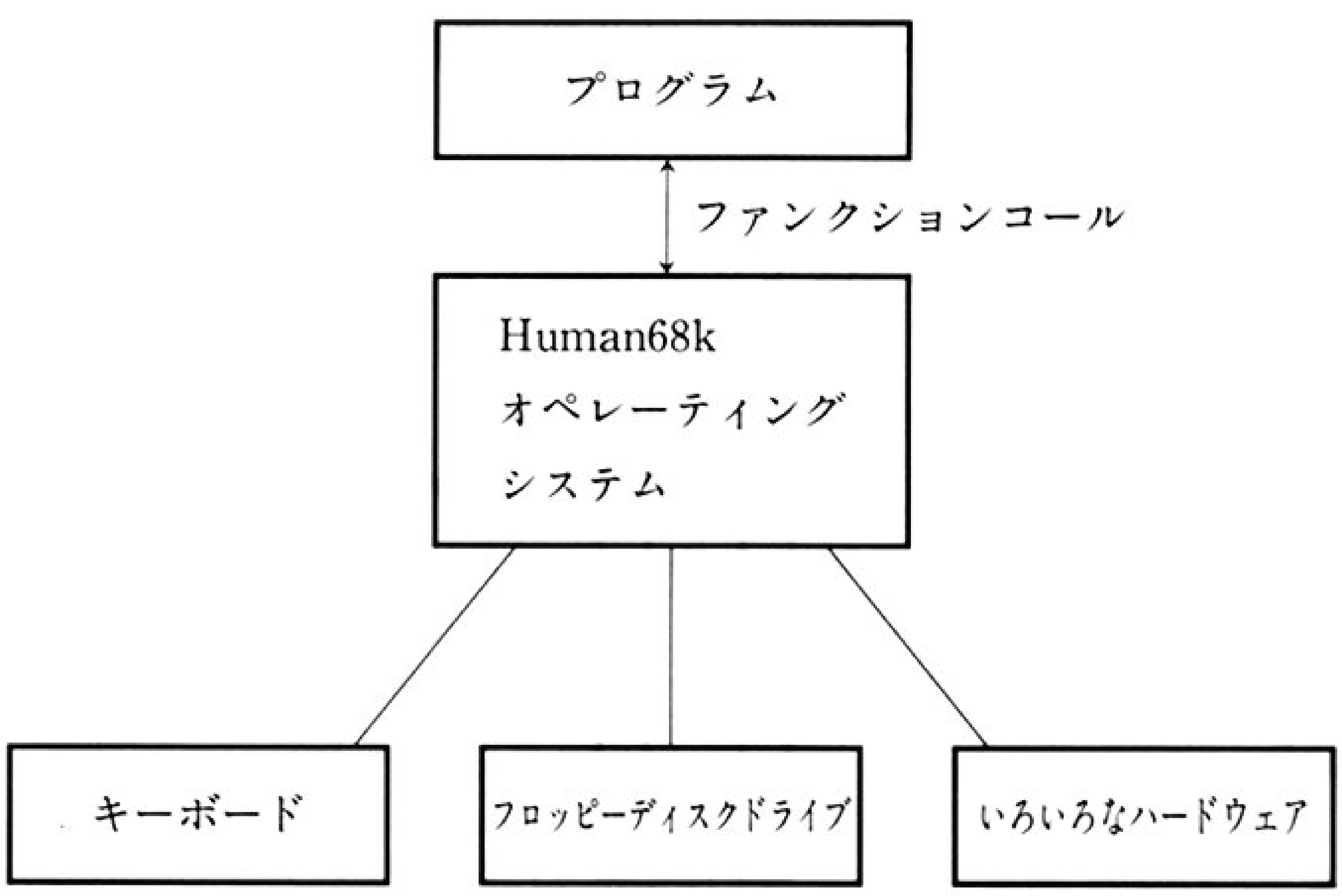


では、「cap」の仕様をまとめてみましょう。

処理……ASCII 小文字コードの ASCII 大文字コードへの変換
入力……ASCII 小文字コードで作成されたデータ
出力……ディスプレイ表示出力

ファンクションコール

さて、次に Human68k といったオペレーティングシステムの上で実行されるプログラムは、オペレーティングシステムの援助を受けることができます。
具体的にその援助の例として、たとえば、キーボードから入力した文字が何かということをオペレーティングシステムに判断させておいて、結果だけをこれから開発するプログラムが受けるようにするということがあります。
キーボードからの 1 文字の入力といっても、X68000 はいろいろな制御を行っているのですが、そういうことは、オペレーティングシステムにまかせてしまおうということです。
このような援助のことを、機能（ファンクション）を呼び出す（コールする）ということから、「ファンクションコール」といいます。
ファンクションコールは、通常アセンブリ言語によるプログラムで扱いますが、C 言語によるプログラムでも使用可能です。



3.1　プログラム仕様の決定

〔備考〕開発キットの中のプログラマーズマニュアルで詳しくファンクションコールの説明をしています。
X68000はユーザーのために、数多くのファンクションコールを用意しています。
また、ファンクションコールには、決まった形式の使用方法がありますが、これらも、そこに示してあります。

「cap」を作成するのには、次のファンクションコールを使用します。

- **キーボードから1文字入力　ファンクションコール8番 getc**
- **ディスプレイへの1文字出力　ファンクションコール2番 putchar**

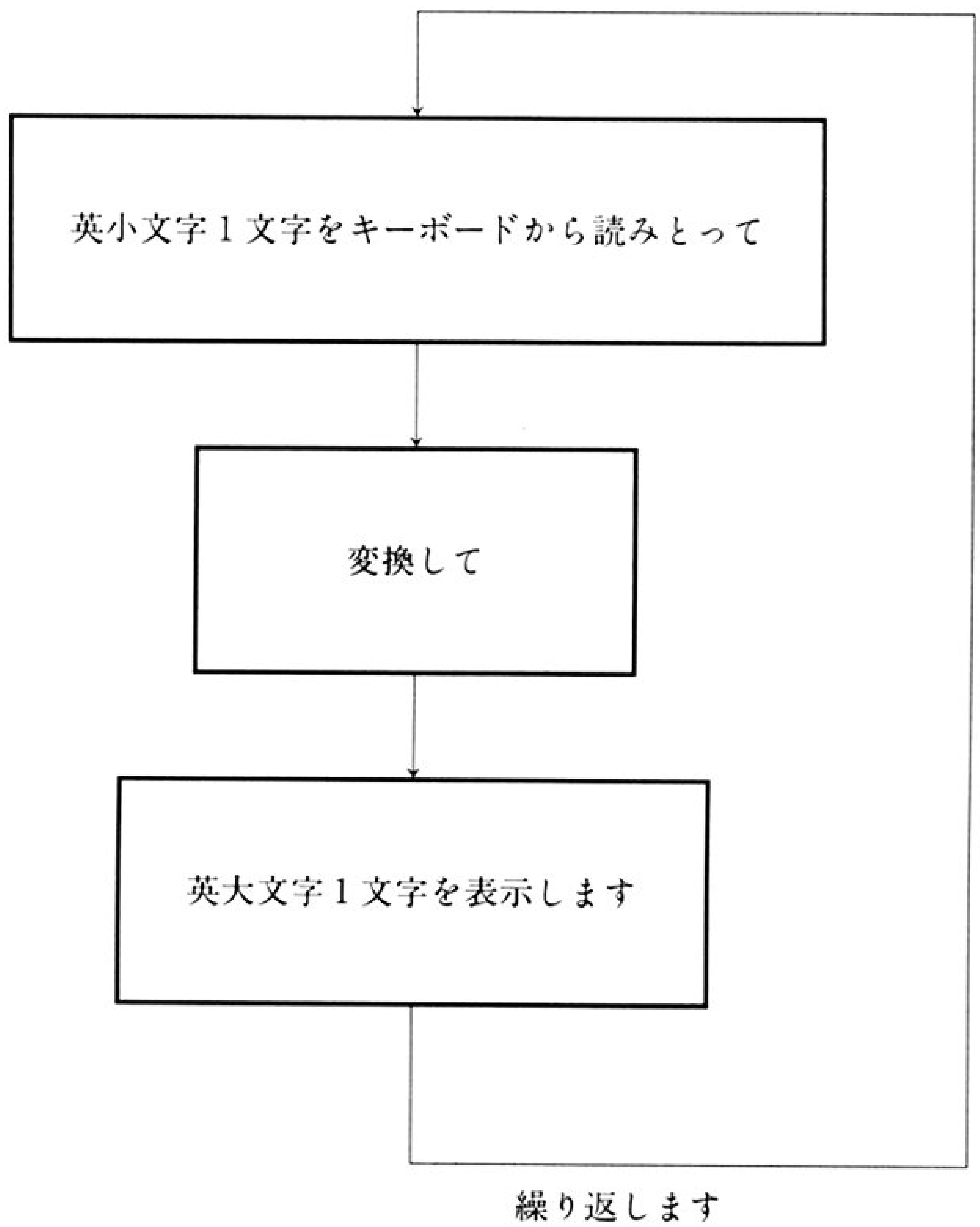
これで、使用するファンクションコールまで決まって、仕様がかたまりました。

処理……ASCII小文字コードのASCII大文字コードへの変換
入力……ASCII小文字コードで作成されたデータ
出力……ディスプレイ表示出力
使用ファンクションコール
　　ファンクションコール08……1文字入力
　　ファンクションコール02……1文字出力

3.1.2 アルゴリズム

プログラムには、初めと終わりがあって、その過程には処理の流れがあります。この処理の流れの手順のことを「手続き」といいます。
どのような手続きをどのような順序で行うかということを具体的に示したものを「アルゴリズム」といいます。
アルゴリズムはプログラムの骨組みともいえます。

まず、



では、この変換はどのようにしたらいいでしょうか? ここが、一番、工夫のいるところです。

3.1　プログラム仕様の決定

キーボードから入力した文字は、コンピュータの内部では、その文字に番号がふられています。
たとえば、ASCIIコードでは、'A'には16進数で41、'a'には16進数で61があてられています。
この番号の対応は次のように決まっています。

英小文字	コード	英大文字	コード
a	61	A	41
b	62	B	42
c	63	C	43
d	64	D	44
e	65	E	45
f	66	F	46
g	67	G	47
h	68	H	48
i	69	I	49
j	6A	J	4A
k	6B	K	4B
l	6C	L	4C
m	6D	M	4D
n	6E	N	4E
o	6F	O	4F
p	70	P	50
q	71	Q	51
r	72	R	52
s	73	S	53
t	74	T	54
u	75	U	55
v	76	V	56
w	77	W	57
x	78	X	58
y	79	Y	59
z	7A	Z	5A

注：コードは16進数です。

a 61 ——→ A 41
b 62 ——→ B 42

この変換の「法則性」は、2進数を使うとはっきりします。
2進数は0と1でビットごとに表現した数字です。
なお、このようなビット単位のプログラム操作にアセンブラプログラムは向いています。

さて、　a は2進数で 0110 0001 です。
　　　　A は2進数で 0100 0001 です。

また、　b は2進数で 0110 0010 です。
　　　　B は2進数で 0100 0010 です。

どうやら、　小文字の a と DF (16 進数) の AND 論理演算を行うと、この変換が表現できそうです。

0 1 1 0 0 0 0 1……61 (小文字の a)
AND
1 1 0 1 1 1 1 1……DF
0 1 0 0 0 0 0 1……41 (大文字の A)

注：このような処理を「マスク」といいます

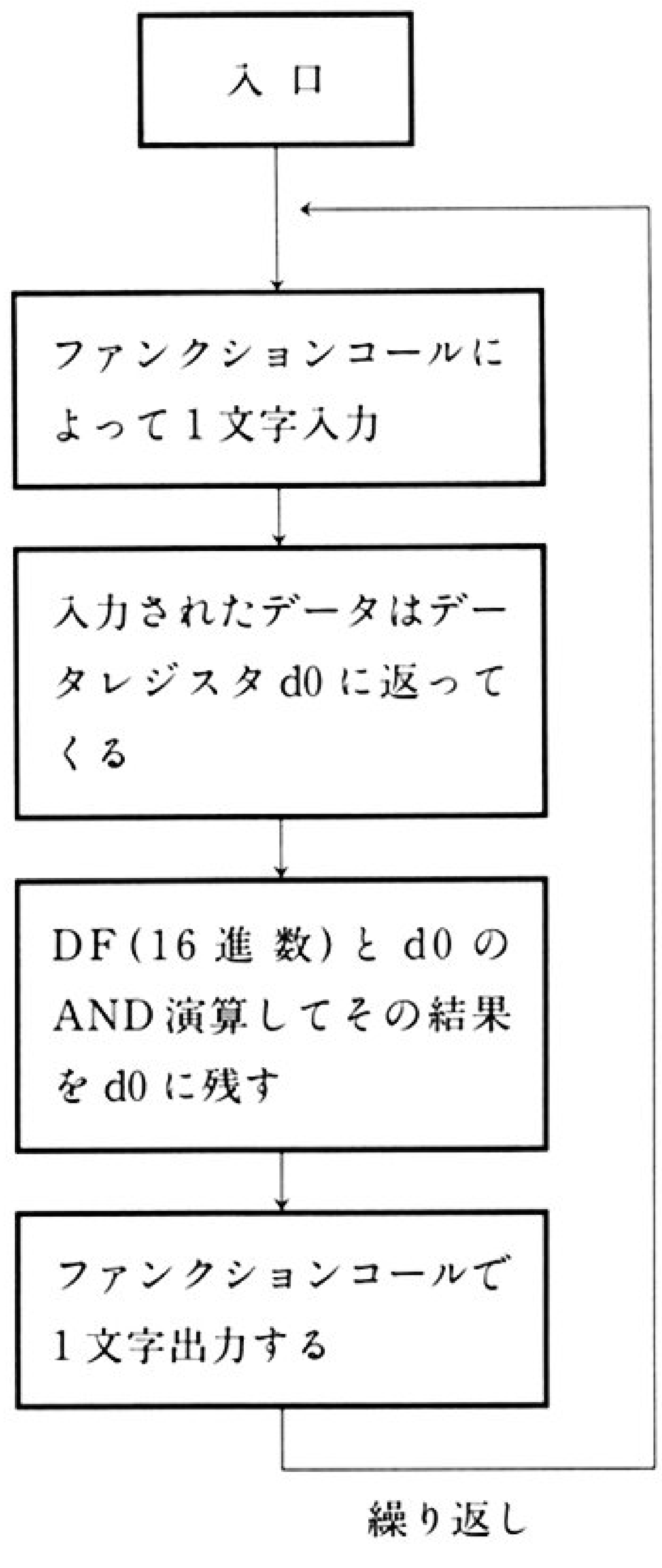
ところで、AND 論理演算においては、双方のビットが1のとき、はじめて1(または真)となります。
論理演算の組み合せとその結果は次のようになります。

A	B	結　果
0	0	0
0	1	0
1	0	0
1	1	1

3.1　プログラム仕様の決定

ところで、このサンプルプログラムには英小文字だけではなく、_（アンダーバー）や：（コロン）などの特殊文字が出てきます。
これらの文字の16進コードが、先の論理演算によって変換されるとどのようになるのでしょうか?　ちょっと気がかりですね。

アルゴリズムをまとめます。



では、MPU68000のインストラクションを参照しながら紙の上にきちんとコーディング（実際のアセンブラプログラムに書き替えること）をしてみましょう。
コーディングの規則は第2部を参照してください。

```
*****************************************
*                                       *
* 小文字を大文字に変換するプログラム      *
* version 1.0                           *
*****************************************
*
* ファンクションコールの定義
*
_getc           equ     $ff08           * 1文字入力
_putchar        eqy     $ff02           * 1文字出力

*
* プログラム本体
*
begin:                                  * 始まり(入口)
                dc.w    _getc           * 1文字入力
                                        * 入力された文字はレジスタd0へ

                and.b   #$df,d0         * $dfとレジスタd0とのAND演算
                                        * 結果はレジスタd0へ

                move.w  d0,-(sp)        * 1文字出力
                dc.w    _putchar        *
                addq.l  #2,sp

                jmp     bigin
*
* プログラム終わり
*
```

3.2 ソースファイルの作成

プログラムをエディタ(ED. X)で書いてソースファイルを作成しましょう。

エディタの起動

例として、エディタをAドライブに置いてソースプログラムをBドライブに作成するとします。

```
B>A : ED cap. s ⏎
```

と入力してください。
エディタが起動します。

コーディングした内容をエディタで書いていきましょう。
エディタの使いかたがわからないときは、HELPキーを押すとヘルプメッセージが出ます。
ヘルプメッセージの8画面目には先ほどのASCIIコードの表示が出ます。
エディタで入力し終えたら、ESCキーとEキーでセーブします。

ファイルがディレクトリにあるか確認してください。

```
B>DIR cap. * ⏎
```

cap. sがありますか?

次に、ソースプログラムの中身も確かめてみます。

```
B>TYPE cap. s ⏎
```

3.2 ソースファイルの作成

画面にソースプログラムが表示されましたか?

```
*****************************************
*                                       *
* 小文字を大文字に変換するプログラム       *
* version 1.0                           *
*****************************************
*
* ファンクションコールの定義
*
_getc           equ     $ff08           * 1文字入力
_putchar        eqy     $ff02           * 1文字出力

*
* プログラム本体
*
begin:                                  * 始まり(入口)
                dc.w    _getc           * 1文字入力
                                        * 入力された文字はレジスタd0へ

                and.b   #$df,d0         * $dfとレジスタd0とのAND演算
                                        * 結果はレジスタd0へ

                move.w  d0,-(sp)        * 1文字出力
                dc.w    _putchar        *
                addq.l  #2,sp

                jmp     bigin
*
* プログラム終わり
*
```

これでソースファイルcap. sが作成できました。

3.3 アセンブラの使用

アセンブラの起動

ソースプログラムができましたので、これをアセンブラ (AS. X) にかけてみます。
例として、アセンブラをAドライブに置き、ソースファイル cap. s をBドライブに置くとします。

起動は次のように行います。

```
B>A:AS cap⏎
```

アセンブラが起動すると、しばらくしてメッセージが表示されます。

```
X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
cap.s                    10:   bad opecode error
cap.s                    23:   undefined symbol error
cap.s                    26:   undefined symbol error
undefined symbol(s)
_putchar
bigin
3 Fatal error(s)
```

どうやら先ほど入力したプログラムにはエラーがあるようです。
そこでエラーの詳細な情報が必要になります。
そのためには、アセンブラのリストファイルを/p スイッチを用いて出力させます。

```
B>A : AS  /p cap⏎
```

すると、ディレクトリにcap. prnというリストファイルができます。
TYPEコマンドでリストファイルを見てみましょう。

```
X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
                                                                 mnt/dd/yy hh:mm:ss
                                                                 Page 1-1


<cap.s>
    1 00000000                    *****************************************
    2 00000000                    *                                       *
    3 00000000                    * 小文字を大文字に変換するプログラム    *
    4 00000000                    * version 1.0                           *
    5 00000000                    *****************************************
    6 00000000                    *
    7 00000000                    * ファンクションコールの定義
    8 00000000                    *
    9 00000000 =0000ff08          _getc           equ     $ff08
cap.s               10:     bad opecode error
   10 00000000                    _putchar        eqy     $ff02
   11 00000000
   12 00000000                    *
   13 00000000                    * プログラム本体
   14 00000000                    *
   15 00000000                    begin:
   16 00000000 ff08                               dc.w    _getc
   17 00000002
   18 00000002
   19 00000002 c03c00df                           and.b   #$df,d0
   20 00000006
   21 00000006
   22 00000006 3f00                               move.w  d0,-(sp)
cap.s               23:     undefined symbol error
   23 00000008                                    dc.w    _putchar
   24 0000000a 548f                               addq.l  #2,sp
   25 0000000c
cap.s               26:     undefined symbol error
   26 0000000c 4ef9                               jmp     bigin
   27 00000012                    *
   28 00000012                    * プログラム終わり
   29 00000012                    *

X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
                                                                 mnt/dd/yy hh:mm:ss
                                                                 Symbols-1



Segment table
 01 text      02 data      03 bss      04 stack

Symbol table
 0000ff08 abs _getc    00000000(01) begin

undefined symbol(s)
_putchar
bigin
3 Fatal error(s)
```

エラーは、文字の書き間違いによるようです。
10 行目に eqy とあるのは equ の誤りです。
そしてこの equ によって_putchar が定義できなかったために 23 行目の _putchar が未定義エラーになっています。
26 行目はラベル begin のつづり字の間違いでした。
つづり字を 2 つ間違えただけで 3 つも関連してアセンブラエラーが出ました。
このようにエラーの原因と表示されるエラーの数は一致しません。
よく注意してください。

では、再び、エディタに戻ってソースファイルの誤りを

eqy ――――――→ equ
jmp bigin ――――→ jmp begin

に直しておきましょう。

再アセンブル

正しくアセンブルされると画面に

```
No Fatal error (s)
```

と表示されます。
リストファイルは次のようになります。

```
X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
                                                            mnt/dd/yy hh:mm:ss
                                                            Page 1-1


<cap.s>
    1 00000000                    ****************************************
    2 00000000                    *                                      *
    3 00000000                    * 小文字を大文字に変換するプログラム   *
    4 00000000                    * version 1.0                          *
    5 00000000                    ****************************************
    6 00000000                    *
    7 00000000                    * ファンクションコールの定義
    8 00000000                    *
    9 00000000 =0000ff08          _getc           equ     $ff08
   10 00000000 =0000ff02          _putchar        equ     $ff02
   11 00000000
   12 00000000                    *
   13 00000000                    * プログラム本体
   14 00000000                    *
   15 00000000                    begin:
   16 00000000 ff08                               dc.w    _getc
   17 00000002
   18 00000002
   19 00000002 c03c00df                           and.b   #$df,d0
   20 00000006
   21 00000006
   22 00000006 3f00                               move.w  d0,-(sp)
   23 00000008 ff02                               dc.w    _putchar
   24 0000000a 548f                               addq.l  #2,sp
   25 0000000c
   26 0000000c 4ef9(01)00000000                   jmp     begin
   27 00000012                    *
   28 00000012                    * プログラム終わり
   29 00000012                    *

X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
                                                            mnt/dd/yy hh:mm:ss
                                                            Symbols-1


Segment table
 01 text      02 data      03 bss       04 stack

Symbol table
 0000ff08 abs _getc     0000ff02 abs _putchar 00000000(01) begin

No Fatal error(s)
```

3.4 リンカの作業

アセンブルが正常に終了すると、

cap. o

というオブジェクトファイルが作成されます。
注：オブジェクトファイルはTYPEコマンドで見ることはできません。

大きなソフトウェアの開発などは、プログラムをいくつかの部分に分割して、それぞれをアセンブルして、複数のオブジェクトファイルを作成します。
そして、これらのオブジェクトファイルをリンカ(LK. X)によって結合します。
ここでは、オブジェクトは1つしかありませんが、リンカの作業は必要です。

リンカの起動

リンカの起動は、この例では

```
B>A : LK cap ⏎
```

で行います。

ただちにリンカが起動して作業が終わると、プロンプト表示状態に戻ります。

```
B>■
```

3.4 リンカの作業

正しくオブジェクトファイルが選択されているか、また、作成された機械語のサイズを知りたいときは/v（バーボースモードスイッチ）をつけます。

```
B>A : LK /v cap ↵
```

すると、次のように作業の状態が表示されます。

```
X68k Linker v2.00 Copyright 1987, 88, 89, 90 SHARP/Hudson
cap. o                                ←——オブジェクトファイル名
text 00000000-00000012(00000012)      ←——機械語サイズ
```

3.5 実　行

リンクが終わると、ディレクトリに

cap. x

という実行可能プログラムが作成されます。

実行してみましょう。
実行は次のように行います。

```
B>cap ⏎
```

さて、キーボードから英小文字を入力してみてください。
[a]と打つと、“A”が表示されます。
[CAPS]キーをオン・オフにしても、それに影響なく英大文字が表示されます。

ところで、このプログラムはどうすると終了するのでしょうか?

[BREAK]キーを押すと終了します。

このプログラムには、実は出口がありませんでした。
通常このようなプログラムを作成することはまれです。
[BREAK]キーを押してこのプログラムが終了したのは、「1 文字入力」のファンクションコールの中でオペレーティングシステムが、[BREAK]キーを理解しているからです。
だから、このようなファンクションコールを使っていれば、[BREAK]キーでプログラムが終了しますが、そうでないと、プログラムが止まらないという悪い事態が発生します。
これが「暴走」とふつう呼ばれているもので、次のようなとき発生することがあります。

3.5 実　行

●プログラムの終了ルーチンが明確でない。
●アルゴリズムが正しくない。

暴走するようなプログラムを作成してはいけませんが、万一のときはリセットするしかありません。

3.6 設計の修正と変更

ここでこのプログラムを次の2つの点から設計の修正と変更を行います。

設計の修正

1文字入力のファンクションコールによらない正常な終了方法を作ります。
終了のためのファンクションコール exit があります。

設計変更

キーボードからの入力でなく、ファイルからの入力ができるようにします。

標準入出力について

このプログラムで使用した1文字入力と1文字出力は標準入出力を対象にしています。
通常、標準入出力は

- 入力がキーボード
- 出力がディスプレイ

になっています。

しかし、この入出力先をオペレーティングシステムのリダイレクションで切り替えることができます。
たとえば、英小文字の文章のファイル text をこれから作成するプログラム“cap”へ入力するには

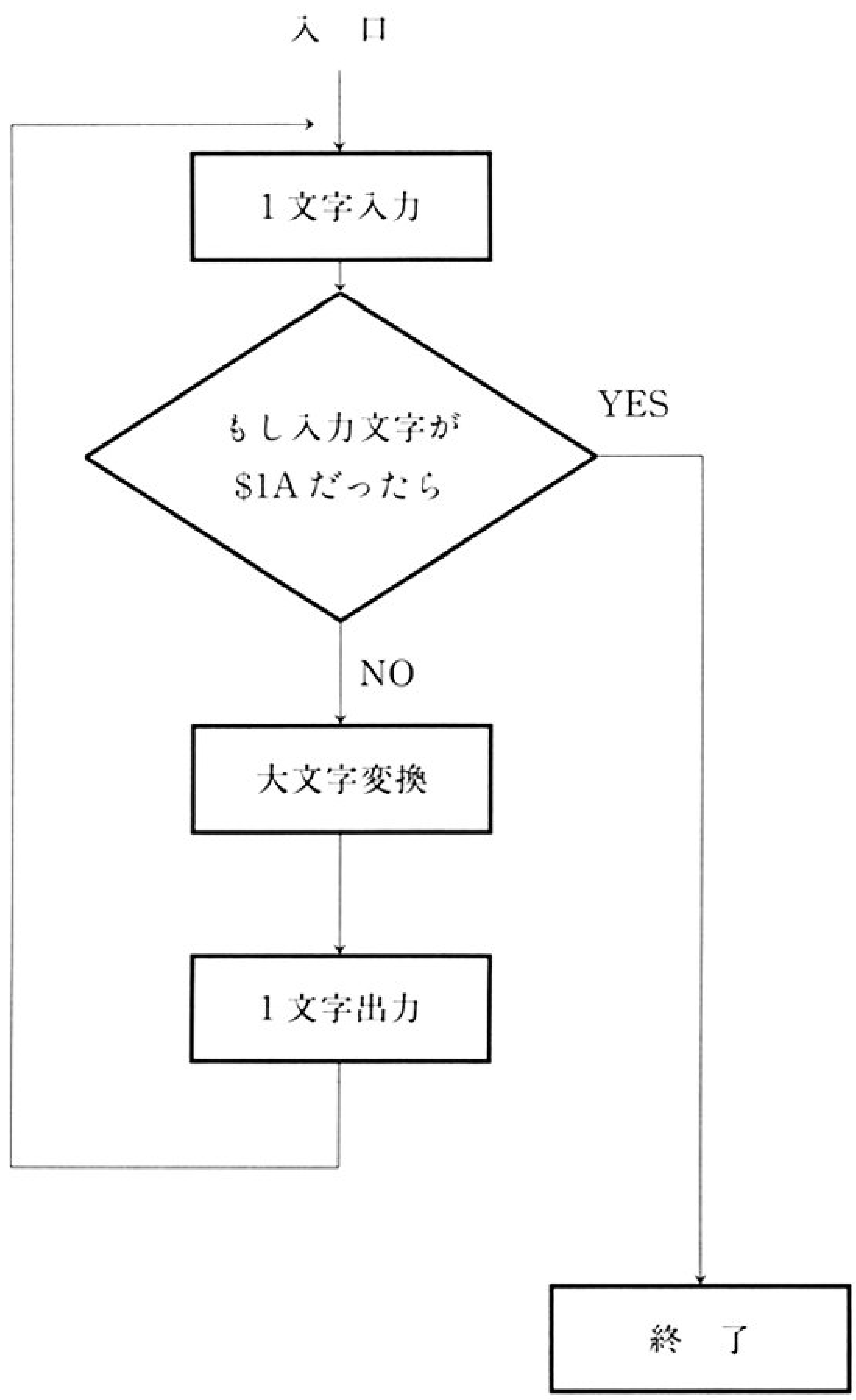
```
B>cap  <  text↵
```

とします。
リダイレクションについては Human68k ユーザーズマニュアルを参照してください。

3.6 設計と修正と変更

さて、文章が含まれるファイルの終わりに$1Aという16進数が終わりのマークとしてあります。
そこで$1Aが入力されると、このプログラムが終了するようにします。

アルゴリズムは次のようになります。



コーディングしてみます。

```
****************************************
*                                      *
* 小文字を大文字に変換するプログラム   *
* version 1.1                          *
****************************************
*
* ファンクションコールの定義
*
_getc           equ     $ff08           * 1文字入力
_putchar        equ     $ff02           * 1文字出力
_exit           equ     $ff00           * 終了

*
* プログラム本体
*
begin:                                  * 始まり(入口)
                dc.w    _getc           * 1文字入力
                                        * 入力された文字はレジスタd0へ

                and.b   #$df,d0         * $dfとレジスタd0とのAND演算
                                        * 結果はレジスタd0へ

                cmp.b   #$1a,d0         * $1aと比較
                beq.b   finish          * 等しいときは、
                                        * finish へ
                                        * それ以外は次へ

                move.w  d0,-(sp)        * 1文字出力
                dc.w    _putchar        *
                addq.l  #2,sp

                jmp     begin           * beginへ戻る

finish:         dc.w    _exit           * 終了
*
* プログラム終わり
*
```

このプログラムを先ほどと同様にアセンブルし、リンクして、実行可能プログラムを作成します。

キーボードから英小文字を入力すると、大文字になります。
終了は$1A という 16 進数をこのプログラムに送るのですが、その方法は、CTRLキーを押しながら、Zのキーを押します。
すると、プログラムが終了してプロンプトへ戻るはずです。

しかし、正しくプロンプトが表示されていません。
現在のところ次のような画面になっているはずです。

```
AAABCQQ      B>
  ↑        ↑   ↑
  |        |   └──ここにプロンプトが出てしまう
  |        |
小文字を入    |
力したとき  ここで[CTRL]キーを押しながら
           [Z]キーを押す。
```

どうやらまだ、完全なプログラムにはなってないようです。

次にエディタで

```
Hello, this is a sample.
Change us into CAP letters.
```

というような小文字を含んだ文章を作り

text

というファイル名でセーブしてください。

さて、先のプログラムで大文字に変換してみましょう。

```
B>cap < text ↵
```

すると

```
HELLO THISISASAMPLE
CHANGEUSINTOCAPLETTERS
```

と表示されます。

カンマ、スペース、ピリオドが不自然です。

ここでも完全なプログラムという感じがしません。

3.7 デバッガ

ここで、次のような文章を作ってtextとしてセーブしてください。

```
This program changes, as follows
        a : A
        b : B
So so good!
```

変換してみます。

```
B>cap  <  text ⏎
```

すると、

```
THISPROGRAMCHANGES ASFOLLOWS
A B>
  └──────→プロンプトに戻る
```

どうやら

a : A (2行目)

のコロンの位置で何か異常な動作が発生しているようです。

なぜこうなったかは、ソースプログラムをよく検討するとわかるのですが、ここではデバッガを使ってこの理由をさぐってみます。

デバッガの起動

デバッガはこの例では

```
B>DB cap. x ⏎
```

と入力すると起動して、次のように表示します。

```
B>DB cap.x⏎
X68k Debugger v2.00 Copyright 1987,88,89,90 SHARP/Hudson
loading cap.x←――――――――プログラム名                CPUの状態 ――┐
No symbol file                                                      │
PC=000AE1E0 USP=0008CD28 SSP=000067F2 SR=0000 X:0  N:0  Z:0  V:0  C:0←┘
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
_GETC
-■
```

プロンプトが

—

というようにバーになっているはずです。

（上記はあくまでも、デバッガ起動の一例であり、表示されるレジスタの値が上記のものと一致するとはかぎりません。）

ここでhを入力すると、ヘルプメッセージが表示されます。

```
-h⏎
A[address]                          : アセンブル
AN[address]                         : アセンブル（ニーモニック表示なし）
B                                   : ブレークポイントの表示
B[bp] address [count]               : ブレークポイントの設定
BC bp                               : ブレークポイントの削除
BD bp                               : ブレークポイントの無効化
BE bp                               : ブレークポイントの有効化
BR                                  : ブレークカウントの初期化
C string                            : コマンドラインの設定
D[size][range]                      : メモリ内容のダンプ
F[size]range data                   : フィルメモリ
G[=address] [address]               : デバッグ中のプログラムの実行
H                                   : オンラインヘルプメッセージ
HC                                  : トレース履歴の消去
HI                                  : トレース履歴の表示
L[range]                            : アセンブリリスト表示
ME[size][address] [data]            : メモリ内容の編集
MEN[size][address]                  : メモリ内容の編集（表示なし）
MM range address                    : メモリ内容の移動
MS[size]range data                  : メモリ内容の検索
```

```
N                               :メモリチェックポイントの表示
N[size][cp] address [cd][data]  :メモリチェックポイントの設定
NC cp                           :メモリチェックポイントの消去
ND cp                           :メモリチェックポイントの無効
NE cp                           :メモリチェックポイントの有効
O                               :画面表示幅の変更
size            s(byte) w(word) l(long)
bp              0-9
cd(condition)   0:<>,1:=
cp              0-9
-- More -- (y/n) ? q
P                               :システムステータスの表示
PS                              :シンボルテーブルの表示
PS symbol                       :シンボルの検索表示
Q                               :デバッガの終了
R filename[,address]            :ファイルの読み込み
R@ address drive record count   :ディスクの物理リード
S[count]                        :ステップ実行
T[=address] [count]             :トレース実行
U[=address] [count]             :表示なしトレース
V                               :コンソールの切り替え
W filename,range                :ファイルの書き込み
W@ address drive record count   :ディスクの物理ライト
X                               :レジスタ内容の表示
X reg                           :レジスタ内容の変更
Y/N                             :ポーズ
Z                               :システム変数の表示
Z num=exp                       :システム変数の設定
? exp                           :16進表示
?? exp                          :10進表示
¥                               :コマンドラインの繰り返し
> filename                      :出力リダイレクト
>> filename                     :出力リダイレクト(アペンド)
>@ filename                     :入力コマンドをファイルに保存
< filename                      :入力リダイレクト
![os_command]                   :OSコマンドの実行
symbol     シンボル名の先頭に.(period)
drive      0:current,1:A,2:B, .....
reg        d0-d7 a0-a7 ssp usp sr ccr pc
operators  + - * / & (and) | (or) ! (not) % (residue) ^ (exor)
number     ??(hex.) .??(symbol) ¥??(dec.) _??(bin.)
-■
```

デバッガにとり込んだプログラムが正しいかどうかを確かめるために、逆アセンブルしてみます。

逆アセンブルというのは、実行可能プログラムからソースプログラムに近い状態の記述に戻すことです。

逆アセンブルはlコマンドを使います。

3.7 デバッガ

```
-l⏎
  000AE1E0        _GETC
  000AE1E2        and.b    #$DF,D0
  000AE1E6        cmp.b    #$1A,D0
  000AE1EA        beq.s    $000AE1F8
  000AE1EC        move.w   D0,-(A7)
  000AE1EE        _PUTCHAR
  000AE1F0        addq.l   #2,A7
  000AE1F2        jmp      $000AE1E0
-l⏎
  000AE1F8        _EXIT──────────── ここまで
  000AE1FA        ori.b    #$00,D0
  000AE1FE         undefined instruction $00FF
  000AE200        ori.b    #$00,D0
  000AE204        ori.b    #$00,D0
  000AE208        ori.b    #$00,D0
  000AE20C        ori.b    #$00,D0
  000AE210        ori.b    #$00,D0
-
```

X68000のデバッガでは、この逆アセンブルリストのように、ファンクションコールがはっきりとわかるように表示されます。

このリストとソースリストを比べてください。

たしかに正しく逆アセンブルされています。

機械語をダンプしてみます。

```
-d
000AE1E0  FF08 C03C 00DF B03C 001A 670C 3F00 FF02       ..タく.°-く..g.?...
000AE1F0  548F 4EF9 000A E1E0 FF00 0000 0000 00FF       T蒐...羚........
000AE200  0000 0000 0000 0000 0000 0000 0000 0000       ................
000AE210  0000 0000 0000 0000 0000 0000 0000 0000       ................
000AE220  0000 0000 0000 0000 0000 0000 0000 0000       ................
000AE230  0000 0000 0000 0000 0000 0000 0000 0000       ................
000AE240  0000 0000 0000 0000 0000 0000 0000 0000       ................
000AE250  0000 0000 0000 0000 0000 0000 0000 0000       ................
-
```

$AE1F8番地に_exitのファンクションコールである$FF00が見えます。

このダンプリストとアセンブラのリストファイルについている機械語と対応させてみてください。

アセンブラリストファイル

```
X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
                                                        mnt/dd/yy hh:mm:ss
                                                        Page 1-1


<cap.s>
    1 00000000                    ****************************************
    2 00000000                    *                                      *
    3 00000000                    * 小文字を大文字に変換するプログラム    *
    4 00000000                    * version 1.1                          *
    5 00000000                    ****************************************
    6 00000000                    *
    7 00000000                    * ファンクションコールの定義
    8 00000000                    *
    9 00000000 =0000ff08          _getc           equ     $ff08
   10 00000000 =0000ff02          _putchar        equ     $ff02
   11 00000000 =0000ff00          _exit           equ     $ff00
   12 00000000
   13 00000000                    *
   14 00000000                    * プログラム本体
   15 00000000                    *
   16 00000000                    begin:
   17 00000000 ff08                               dc.w    _getc
   18 00000002
   19 00000002
   20 00000002 c03c00df                           and.b   #$df,d0
   21 00000006
   22 00000006
   23 00000006 b03c001a                           cmp.b   #$1a,d0
   24 0000000a 670c_00000018                      beq.b   finish
   25 0000000c
   26 0000000c
   27 0000000c
   28 0000000c 3f00                               move.w  d0,-(sp)
   29 0000000e ff02                               dc.w    _putchar
   30 00000010 548f                               addq.l  #2,sp
   31 00000012
   32 00000012 4ef9(01)00000000                   jmp     begin
   33 00000018
   34 00000018 ff00               finish:         dc.w    _exit
   35 0000001a                    *
   36 0000001a                    * プログラム終わり
   37 0000001a                    *

X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
                                                        mnt/dd/yy hh:mm:ss
                                                        Symbols-1


Segment table
 01 text      02 data      03 bss      04 stack

Symbol table
 0000ff00 abs _exit    0000ff08 abs _getc    0000ff02 abs _putchar 00000000(01)
begin    00000018(01) finish

No Fatal error(s)
```

ここで、qを入力します。

−q

デバッガを抜けます。

トレース

デバッグの基本はトレースです。
トレースというのは、プログラムをソースプログラムの1行（行をここではステップといいます）ごとに実行してプログラムの処理状況を見てゆくことです。

再び

```
B>DB cap. x ⏎
```

でデバッガに入ります。

_GETC

という、表示の次に

-t ⏎

と、tコマンドを入力してください。
するとデータの入力を求めてきますので、"a"を入力してください。

a ←———"a"を入力。画面には表示されません。

```
-t ⏎
PC=000AE1E2 USP=0008CD28 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000061 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
and.b    #$DF,D0
                        ―"a"のコード61(16進数)がレジスタD0に入力されました。
```

次をトレースします。

```
-t ⏎
PC=000AE1E6 USP=0008CD28 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000041 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
cmp.b    #$1A,D0          ―ANDのために41(16進数)に変化しました。

-t ⏎
PC=000AE1EA USP=0008CD28 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000041 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
beq.s    $000AE1F8

-t ⏎
PC=000AE1EC USP=0008CD28 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000041 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
move.w   D0,-(A7)                    ;0008CD26(0000)

-t ⏎
PC=000AE1EE USP=0008CD26 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000041 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD26
_PUTCHAR
                  ―出力後0になりました。
-t ⏎
APC=000AE1F0 USP=0008CD26 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD26
addq.l   #2,A7

-t ⏎
PC=000AE1F2 USP=0008CD28 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
jmp      $000AE1E0                   ;000AE1E0(FF08C03C)

-t ⏎
PC=000AE1E0 USP=0008CD28 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
_GETC
```

再び、_GETCになったところでtコマンドを入力すると、データの入力を求めてきますので問題のおこった":"(コロン)を入力してみます。

```
-t⏎
PC=000AE1E2 USP=0008CD28 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  0000003A 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
and.b   #$DF,D0
```

3A(16進数)はコロンのコードです。

トレースを続けます。

```
-t⏎
PC=000AE1E6 USP=0008CD28 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  0000001A 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
cmp.b   #$1A,D0
```

AND後に1A(16進数)になってしまいました。

コロンのコード$3Aを$DFでAND演算をすると$1Aになります。
$1Aはファイルの終わりを示すマークと同じ値になってしまいました。

トレースを続けます。

```
-t⏎
PC=000AE1EA USP=0008CD28 SSP=000067F2 SR=8004 X:0  N:0  Z:1  V:0  C:0
D  0000001A 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
beq.s   $000AE1F8
-t⏎
PC=000AE1F8 USP=0008CD28 SSP=000067F2 SR=8004 X:0  N:0  Z:1  V:0  C:0
D  0000001A 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AE0E0 000AE1FA 0008D638 000851C0  000AE1E0 00000000 00000000 0008CD28
_EXIT
-t⏎
program terminated normally
```

注意（1A → beq.s $000AE1F8）

注意（1A → _EXIT）

どうやらこのプログラムはこの1A (16進)をファイルの終わりとみなしてプログラムを終了させてしまいました。

デバッグの結果

デバッガによってトレースを行っていくと、このプログラムには大きな問題があることがわかりました。

また、設計に戻ってやり直さなくてはなりません。
どういうアルゴリズムにしたらよいでしょうか?

この章では、プログラム開発手順の例ということなので、このプログラムの開発についてはここで中断します。

3.8 まとめ

ここまでのプログラム開発手順の例で、もっとも基本的な「ソフトウェア開発キット」の使用法はご理解いただけたと思います。

アルゴリズムを練ってすばらしいプログラムを作成し、X68000 の機能のすべてを引き出してください。

第 4 章

アセンブラ

アセンブラが使用するファイル
使用書式と起動方法
アセンブラのスイッチ
アセンブラのエラーメッセージ

アセンブラ XAssembler は、X68000 のマイクロプロセッサ用マクロアセンブラ言語です。

XAssembler は、アセンブラで書かれたソースファイルをアセンブルし、オブジェクトファイルを生成します。
このオブジェクトファイルを XLinker でリンクすると、実行可能なプログラムとなります。

4.1 アセンブラが使用するファイル

アセンブラはアセンブルにあたって、以下のファイルを使用します。
ただし、以下のファイルのうちユーザーが直接使用するファイルは、ソースファイルとオブジェクトファイルだけです。

4.1.1 入力ファイル

入力ファイルには、ソースファイルとインクルードファイルの2種類があります。

●ソースファイル

アセンブリ言語で記述されたファイルです。
アセンブラの記述方法、構文などについては、第2部リファレンスマニュアルの「第1章　アセンブラ」を参照してください。

●インクルードファイル

これもソースファイルですが、各プログラム間で共通に使用するシンボルなどを定義するときに使用します。
インクルードファイルを使用するときは、アセンブラのソースプログラム中に.include擬似命令を記述するとともに、アセンブラのコマンドラインに、/iスイッチを指定します。
これにより、インクルードファイルのソースプログラムがソースプログラム中に挿入されるので、プログラマはコーディングの手間が省けるとともに、プログラム開発の生産性が向上します。

注：スイッチは／(スラッシュ）で指定するのが原則ですが、－（ハイフン）も使用できます。

4.1.2 テンポラリファイル

これは、アセンブル時の作業用ファイルです。
アセンブル時の作業領域は主記憶上に確保されますが、これが不足した場

合、ディスク上にテンポラリファイルが自動的に作成されます。
アセンブル時には、アセンブラによりテンポラリファイルが作成されるので、アセンブルが終了するまでフロッピーディスクをとりはずしてはいけません。
アセンブルが終了すると、テンポラリファイルは自動的に削除されます。
なお、テンポラリファイルのパス名はシステム変数tempより参照されます。
ただし、/tスイッチで指定された場合は、こちらの指定が優先されます。

4.1.3 出力ファイル

出力ファイルには、オブジェクトファイルとリストファイルの2種類があります。

●オブジェクトファイル

アセンブル処理によって作成されたオブジェクトプログラムを格納するファイルです。
オブジェクトファイル名は、／oスイッチで指定します。
指定がなければ、アセンブラはソースファイルの拡張子sをoとしたファイルを自動的に作成します。

●リストファイル

アセンブルリストを格納するファイルです。
リストファイル名は、／pスイッチで指定します。
／pスイッチのみを指定して、リストファイル名を指定しないと、アセンブラはソースファイル名の拡張子sをprnとしたファイルを自動的に作成します。

4.2 使用書式と起動方法

起動前に確認すること

- アセンブラが使用するファイルが、ディスク上、および処理を行うパス上にありますか?
- パスの中にファイルや、アセンブラがテンポラリファイルを作成するのに十分な空き領域がありますか?

アセンブラの起動方法と書式

コマンドラインから、次のような書式で入力します。

```
AS [<スイッチ>] <ソースファイル名>↵
```

4.3 アセンブラのスイッチ

アセンブラ実行時に、アセンブル時の動作を制御したり、アセンブラに必要な情報を与えるために、コマンドライン上に指定する'／'に続く英字1文字をスイッチといいます。

```
AS /t¥tmp /oobjout source ⏎
   ↑      ↑
   └──────┴──────────スイッチ
```

アセンブラには以下のようなスイッチがあります。

スイッチ	機　能
／8	シンボルの有効データ長の指定
／a	絶対ショートアドレス形式対応モードの指定
／d	全シンボルの外部エントリ指定
／i	インクルードファイルのパス指定
／m	シンボルの最大個数の指定
／n	最適化の禁止
／o	オブジェクトファイル名の指定
／p	リストファイル名の指定
／s	シンボルの定義
／t	テンポラリファイルのパス指定
／u	未定義シンボルの外部参照指定
／w	警告メッセージの出力禁止
／x	シンボル情報の出力ファイルの指定

次にスイッチごとに説明します。
なお、書式の説明にあたっては以下の表記を使用します。

＜　＞：指定が必要であることを示します。
［　］：指定が省略可能であることを示します。

/8　シンボルの有効データ長の指定

書式　／8

機能　シンボルの有効データ長を8文字とします。

解説　シンボルの有効データ長を8文字までとします。
この結果8文字を超えるシンボルを定義しても、アセンブラは8文字までしか識別しません。

例　／8スイッチの使用例を以下に示します。

```
AS　／8　／i¥include source ⏎
```

上記のように指定した場合、以下の①②は同一シンボル、③は異なるシンボルとして扱われます。

```
① ABCDEFGJHIJK :
② ABCDEFGJH123 :
③ ABCDEFG68332 :
```

／8スイッチを指定しなければ、①②③はすべて異なるシンボルとして扱われます。

／a 絶対ショートアドレス形式対応モードの指定

書　式　／a

機　能　絶対ショートアドレス形式のコードを出力する。

解　説　ソースファイル中で絶対アドレス参照をしている命令が記述されていて、かつその絶対アドレスが$00000000～$00007FFF または$FFFF8000～$FFFFFFFF の範囲内の場合、このスイッチが指定されていると、その命令は、絶対ショートアドレス形式と認識されます。
／a スイッチを指定しないと、絶対ショートアドレス形式のコードを生成できる場合でも、それは絶対ロングアドレス形式になります。

例　／a スイッチの使用例を以下に示します。

```
AS　／a　source↵
```

ソースファイル source. s の中で絶対ショートアドレス形式の命令が存在するなら、絶対ショートアドレス形式のコードを生成します。

/d　全シンボルの外部エントリ指定

書　式　/d

機　能　全シンボルを外部エントリとします。

解　説　本スイッチは、アセンブル時にシンボルがあると、これをすべて外部エントリとします。

たとえば、他のプログラムから呼ばれるシンボルが該当プログラム内に多数存在する場合、これを個別に擬似命令で定義する代わりに、本スイッチで代行、すなわちこれらのシンボルをすべて外部エントリ扱いとすることができます。

例　/dスイッチの使用例を以下に示します。

```
AS  /d  source⏎
```

/dスイッチを指定した場合、以下のラベルはすべて外部エントリ扱いとなります。

プログラムA

```
LABEL1 :
LABEL2 :
LABEL3 :
```

上記のケースは、/dスイッチを指定せず、プログラム中で以下のように指定した場合と同じです。

```
XDEF LABEL1, LABEL2, LABEL3

LABEL1 :
LABEL2 :
LABEL3 :
```

/i インクルードファイルのパス指定

書　式　／i<パス名>

機　能　インクルードファイルのパスを指定します。

解　説　アセンブル時のインクルードファイル挿入に必要となるパスを指定します。
本スイッチはインクルードファイルが、ソースファイルとは異なるディレクトリに存在する場合に使用することができます（インクルードファイルとソースファイルのディレクトリを分けるのは、ファイルの管理を容易にするためです）。

例　／i スイッチの使用例を以下に示します。

```
AS　／i¥include source ⏎
```

ソースファイル名 source. s とインクルードファイルのパス名¥include でアセンブラを起動します。

このときソースファイル source. s ではインクルードファイル名を以下のように指定します。

ソースファイル　source. s

```
      :
.include  FILE1
      :     ↑
      :     └─ インクルード
      :         ファイル名
.include  FILE2
```

インクルードファイル FILE1, FILE2 はいずれもソースファイルとは異なるディレクトリ、すなわち¥include 上にあるので、アセンブラ起動時に／i スイッチの指定を省略するとエラーになります。

/m　シンボルの最大個数の指定

書式　/m<nn>

機能　シンボルの最大個数を指定します。

解説　定義可能なシンボルの最大個数を指定します。
指定可能な数値の範囲は 271 以上 32768 未満です。
デフォルトは 2000 個です。
なお、アセンブラ実行時におけるシンボル 1 個あたりのメモリ占有容量は、約 28 バイトです。

例　/m スイッチの使用例を以下に示します。

```
AS  /m4000  /u  /i¥include source ⏎
```

使用できるシンボルの個数は 4000 となります。

/n 最適化の禁止

書式　/n

機能　前方参照の最適化を禁止します。

解説　アセンブル時の前方参照の最適化を制御します。
なお、最適化の対象となる命令は、BRA、BSR、Bcc の 3 つです。

例　/n スイッチの使用例を以下に示します。

```
AS  /n source ↵
```

/n スイッチの指定があると、前方参照の最適化は行われません（以下の例では、ディスプレイスメントとして、BSR 命令には第 2 ワードも作成されます。

```
                    .
                    .
                    .
00005030 61000062_00005094        bsr      label
00005034 284A                     movea.l  A2,A4
                    .
                    .
                    .
00005094                     label:
00005094 4E40                     trap     #0
```

上記のように、/n スイッチの指定がないと、前方参照の最適化が行われます（BSR 命令のディスプレイスメントは 8 ビットとなり、第 2 ワードは作成されません）。

```
                    .
                    .
                    .
00005030 6160_00005092            bsr      label
00005034 284A                     movea.l  A2,A4
                    .
                    .
                    .
00005092                     label:
00005092 4E40                     trap     #0
```

/o　オブジェクトファイル名の指定

書式　／o＜ファイル名＞

機能　オブジェクトファイルの名前を指定します。

解説　アセンブル時に生成されるオブジェクトファイルの名前を指定します。
指定しないと、ソースファイル名の拡張子 s を o としてオブジェクトファイルが自動的に生成されます。

例　／o スイッチの使用例を以下に示します。

```
AS  /oobjout source ↵
```

オブジェクトファイル名は objout. o となります。

```
AS source ↵ (o スイッチを指定しない場合)
```

オブジェクトファイル名は source. o となります。

/p リストファイル名の指定

書式 /p［<ファイル名>］

機能 アセンブルリストの出力ファイル名を指定します。

解説 アセンブルリスト出力ファイル名を指定します。
/pスイッチのみを指定して、リストファイル名を指定しないと、アセンブラは自動的にリストファイルを作成します。
この場合ファイル名は、ソースファイル名の拡張子sをprnに置き換えたものとなります。

例 /pスイッチの使用例を以下に示します。

```
AS /oobject /psysout source ⏎
```

上記の例では、アセンブリリストの出力ファイル名はsysoutとなります。

```
AS /p /oobject source ⏎
```

この例では、アセンブリリストの出力ファイル名はsource. prnとなります。

/S シンボルの定義

書式　/s<シンボル名>

機能　シンボルを定義します。

解説　シンボルを定義します。

例　/s スイッチの使用例を以下に示します。

```
AS  /smsglng  /8 source ⏎
```

シンボル 'msglng' が定義されます。

/t テンポラリファイルのパス指定

書式 /t<パス名>

機能 テンポラリファイルが作られるパスを指定します。

解説 アセンブル時に使用するテンポラリファイルが作られるパスを指定します。
テンポラリファイルのパス名のデフォルトは、システム変数 TEMP に設定されたパス名となりますが、/t スイッチで指定された場合には、こちらが優先されます。
本スイッチは、システム変数 TEMP に設定されたディレクトリやカレントディレクトリにテンポラリファイルが作成できない場合（ファイル領域不足によって）に指定することができます。

例 /t スイッチの使用例を以下に示します。

```
AS  /t¥tmp source ⏎
```

ソースファイル名 source. s、テンポラリファイルのパス名¥tmp でアセンブラを起動します。

/u　未定義シンボルの外部参照指定

書　　式　／u

機　　能　未定義シンボルを外部参照とします。

解　　説　未定義シンボルを外部参照とします。
このスイッチは、該当シンボルを定義しているモジュールがまだ作成されていないときに、このシンボルをすべて外部参照とします。

例　／u スイッチの使用例を以下に示します。

ソースファイル soruce. s において、'test'というシンボルが定義されずに参照されている場合、

```
AS source ↵
```

とアセンブルすると、以下のように"未定義シンボルが存在する"という意味のエラーメッセージが出力されて止まります。

```
source. s        7 : undefined symbol error
undefined symbol (s)
test
1 Fatal error (s)
```

このような場合、／u スイッチをつけることで、アセンブルできます。

```
AS  /u source ↵
```

なお、source. s 中で、以下のようにシンボル'test'を外部参照シンボルとして宣言しておくと、／u スイッチなしでもエラーは発生しません。

```
xref test
```

／W 警告メッセージの出力禁止

書式 ／w

機能 警告メッセージの出力を禁止します。

解説 アセンブル時にエラーが発生しても、それが致命的なものでなければ警告メッセージが出力されますが、この警告メッセージを出さないようにしたい場合にこのスイッチを使用します（なおこのとき、警告メッセージを無視してリンクを行い、実行可能ファイルを作成しても、通常は問題ありません）。

例 例えば、以下に示すようにソースファイル source. s をアセンブルしたとしましょう。

```
AS source ↵
```

もし、source. s に記述されているアセンブリ言語の中で、絶対アドレス参照をしている命令が存在した場合、以下ような警告メッセージが表示されます。

```
source. s        6 : Warning : absolute addressing
        move.l 0, d0
```

しかし、以下のように source. s を／w スイッチをつけてアセンブルすると上記の警告メッセージは出力されません。

```
AS  ／w source ↵
```

/x　シンボル情報の出力ファイルの指定

書式　/x [<ファイル名>]

機能　アセンブル時に使用したシンボル情報の出力ファイルの指定

解説　シンボル情報の出力ファイル名を指定します。
/x スイッチのみを指定して、ファイル名を指定しなかった場合、シンボル情報を画面に出力します。
ファイル名を指定すると、そのファイルを作成してシンボル情報を格納します。

例　/x スイッチの使用例を以下に示します。

```
AS  /x source ⏎
```

ソースファイル source. s をアセンブルし、そのシンボル情報を画面に表示します。

```
AS  /x source. sym source ⏎
```

ソースファイル source. s をアセンブルし、そのシンボル情報を source. sym に格納します。

4.4 アセンブラのエラーメッセージ

エラーメッセージ	意　味
Abort：Device full	ディスク容量に空きが無い＊
Abort：file read error	読み込みエラー＊
bad opecode error	オペコードが存在しないか、不適当です
division by zere error	0による除算です
expression error	式が不適当です
feature not available error	サポートされていない表記です
file not found error	ファイルが見あたりません
file open error	ファイルのオープンができない＊
file write error	書き込みエラー＊
forced error by fail directive	制御命令による強制的なエラー
illegal addressing error	命令のアドレス形式が不適当です
illegal operand error	オペランドが不適当です
illegal quick size error	クイックで使用できない値が使われた
illegal relative error	命令で認められていないアドレス形式です
illegal size error	命令のサイズが不適当です
illegal shift count error	シフト数が不適当です
illegal value error	式の結果が不適当です
line too long error	行が長すぎます
macro nesting over error	マクロのネスティングが深すぎます
missing macro error	マクロのネストが不適当です
missing if error	条件つきアセンブリのネストが不適当です
no symbol error	ステートメントにシンボルが必要です

＊のエラーはアセンブルの実行を途中で中止します。

4.4　アセンブラのエラーメッセージ

エラーメッセージ	意　味
not enough memory	メモリー領域が不足しています＊
overflow error	オーバーフローです
redefinition error	シンボルの再定義です
register error	レジスタが不適当です
register size error	レジスタのサイズが不適当です
symbol not defined error	定義をもたないシンボルが使用されています
temporary file open error	テンポラリーファイルが作れません＊
too many include file error	インクルードファイルのネストが深過ぎます
too many symbols error	シンボルが多過ぎます
undefined symbol error	シンボルが未定義でかつグローバル宣言もされていません

＊のエラーはアセンブルの実行を途中で中止します。

エラーメッセージの出力例

```
test. s          1 :   undefined symbol error
  ↑              ↑        ↑
ファイル名       行番号   エラーメッセージ
```

第5章

リンカ

リンカとは
リンカが使用するファイル
使用書式と起動方法
リンカのスイッチ
LK エラーメッセージ一覧

この章ではリンカ XLinker の使用法を説明します。
リンクを開始する前に必ず読んでください。

5.1 リンカとは

アセンブルしたオブジェクトをリンクして実行可能プログラムを作成するには、リンカXLinkerを使います。
XLinkerの機能は、次の通りです。

- 別々にアセンブルしたオブジェクトを結合します。
- 未定義の外部参照をライブラリファイル、アーカイブファイルから探し、外部参照を解決します。
- 外部参照の解決やエラーメッセージを示すリストを作成します。

高級言語であるCなどのコンパイラで作成したアセンブラのソースや、初めからアセンブラで作成したソースは、いったん、アセンブルされると、同一に扱うことのできるオブジェクトになります。
通常、開発はプログラムを複数のモジュールに分割して作成、アセンブル、その後にまとめ上げるという手法をとります。
たとえば、C言語でプログラムの大半、特にアルゴリズムの中心部分を作成し、細かな制御が必要なI/Oに関連する部分をアセンブラで作成し、これらを組み合せる場合があります。
このようなオブジェクトファイルをまとめて1つの実行可能プログラムを作成することがリンカの働きです。

また、C言語などでは、関数の多くが、オブジェクトの形式でライブラリとしてサポートされているので、そのためにもリンカが必要です。
リンカは、分割されている複数のオブジェクトをまとめ上げます。
このとき、別のオブジェクトの一部を利用するルーチンを参照したり、ライブラリ関数を参照するときは、リンカは、まず、オブジェクトを配置して1つにまとめ上げ、次に外部の参照の関係を正しくします。

5.2 リンカが使用するファイル

リンカは、入力ファイル、出力ファイルおよびマップファイル、テンポラリファイルを使用します。

入力ファイル

種類	省略時の拡張子	作成経緯
オブジェクト	o	コンパイラまたは アセンブラ
アーカイブ	a	アーカイバ
ライブラリ	l	ライブラリアン

出力ファイル

種類	省略時の拡張子	使用目的
実行ファイル	x	実行可能プログラム
マップファイル	map	参考資料

●マップファイル

／p スイッチを指定することによってマップファイルを出力することができます。

マップファイルには、その実行可能プログラムのオフセット値、セグメントタイプ、シンボル名が格納されます。

●テンポラリファイル

リンカは、実行可能プログラムの作成に使用するデータの保持に主記憶領域を使います。

ここで処理するオブジェクトが大きくなればなるほど、主記憶領域を多く必要とします。

そこでディスクにテンポラリファイルを自動的に作成します。

このようにリンカは、テンポラリファイルを作成するので、リンカの処理が終了するまで、テンポラリファイルが作成されるドライブは常にアクセス可能な状態でなければなりません。

5.2 リンカが使用するファイル

リンクが終了すると、テンポラリファイルは自動的に消去されます。
通常の使用では、このファイルは必ず作成されるので、テンポラリファイルが作成されるドライブには十分な空き領域が存在しなければなりません。
そこでテンポラリファイルを作成するためのパス（テンポラリパス）を指定しておくことができます。

例

```
LK ／t b:¥temp sample ⏎
   └テンポラリファイルのパス指定
```

／tスイッチによってテンポラリパスを指定しないときは、環境変数TEMPを参照します。
TEMPが設定されていても、／tスイッチによる指定がしてあれば、スイッチの指定を優先します。

アーカイブファイル（拡張子がaのファイル）を高速にリンクしたい場合は、次のようにファイルを変換してください。

例

```
LIB sample. l sample. a ⏎
              └アーカイブファイル名
```

5.3 使用書式と起動方法

起動前に確認すること

- ●リンカが使用するファイルがディスク上、および処理を行うパス上にありますか?
- ●パスの中にファイルやリンカがテンポラリファイルを作成するのに十分な空き領域がありますか?

リンカの起動方法と書式

コマンドラインで次のような書式で入力します。
オブジェクトファイル名は、複数個指定できます。
また、オブジェクトファイル名の他に、オブジェクトをまとめたライブラリファイル名も指定できます。
ライブラリファイルを指定すると、リンカは自動的にその中に含まれるオブジェクトを探します。

LK [スイッチ] <オブジェクトファイル名> [<オブジェクトファイル名>……]

オブジェクトファイル名は、複数個指定できます。
オブジェクトファイル名の代わりに、オブジェクトをまとめたアーカイブファイル名やライブラリファイル名も指定できます。

正しくコマンドラインが入力されると、ただちにリンカが起動します。
もし誤ったスイッチを指定すると次のようなヘルプメッセージが表示されます。
コマンドラインを確認し直してください。

```
X68k Linker v2.00 Copyright 1987,88,89,90 SHARP/Hudson
使用法：lk ［スイッチ］ ファイル ［ファイル・・・］
          /m nn           最大シンボル数(201<nn<65536)
          /t path         テンポラリパス指定
          /o file         オブジェクトファイル名
          /b address      ベースアドレス指定
          /i file         インダイレクトファイルの指定
          /v              バーボーズモード
          /x              シンボルテーブルの出力禁止
          /p              マップファイルの出力
```

また、指定したオブジェクトファイルがみつからないときは、

```
file open error（ファイル名）
```

と、表示されます。

ディレクトリまたは、ライブラリファイル、アーカイブファイルを確認して下さい。

5.4 リンカのスイッチ

リンカのスイッチは、リンクエディット時の動作を制御したり、リンクエディットに必要な情報を与えるために使用します。

リンカには以下のようなスイッチがあります。

スイッチ	機　能
／b	ベースアドレスの指定
／i	インダイレクトファイルの指定
／m	シンボルの最大個数の指定
／o	実行可能プログラムのファイル名の指定
／p	マップファイルの出力
／t	テンポラリファイルのパス指定
／v	バーボーズモード
／x	シンボルテーブルの出力禁止

次ページ以降で、各スイッチごとに説明します。
なお、書式の説明にあたっては以下の表記を使用します。

＜　＞：指定が必要であることを示します。
［　］：指定が省略可能であることを示します。

/b　ベースアドレスの指定

書　　式　/b<ベースアドレス>

機　　能　ベースアドレスを指定します。

解　　説　実行可能プログラムをメモリ上にローディングするときのベースアドレス、すなわちロード開始の基点となるアドレスを指定します。
ベースアドレスは16進数で指定します。

例　/bスイッチの使用例を以下に示します。

```
LK  /b 60000 object1 object2 object3
```

/i　インダイレクトファイルの指定

書式　／i<ファイル名>

機能　インダイレクトファイルを指定します。

解説　リンク時のインダイレクトファイルを指定します。
インダイレクトファイルには、リンク時に使用するスイッチとオブジェクトファイル名を記述します。
この指定は、スイッチやオブジェクトファイルの数が多く、コマンドライン上で長くなる場合や同じ指定を何回も行うような場合に効果的です。
インダイレクトファイルが指定されると、リンカはその内容を参照してリンクを行います。
一度インダイレクトファイルを作成しておけば、リンクのたびにスイッチやオブジェクトファイルを指定する必要がないので、リンクの作業効率を向上させることができます。
なお、インダイレクトファイルの作成はエディタ(ED. X)によって行います。

例　／i スイッチの使用例を以下に示します。

たとえば、

```
LK  /m 4000  /t b:temppath  /o ldmdl object1 object2 object3
object4 object5 ⏎
```

と入力するところを、

インダイレクトファイル indirect を作成し、これに、上記のスイッチとオブジェクトファイル名を登録しておけば、

```
LK  /i indirect ⏎
```

と入力するだけで済みます。

インダイレクトファイル indirect の内容

```
/m 4000
/t b:temppath
/o ldmdl
object1
object2
object3
object4
object5
```

/m シンボルの最大個数の指定

書式　／m<nn>

機能　シンボルの最大個数を指定します。

解説　定義可能なシンボルの最大個数を指定します。
指定可能なシンボルの個数は以下の範囲です。

201<nn<65536

なお、デフォルトは2000個です。

例　／m スイッチの使用例を以下に示します。

```
LK  /m 4000 object1 object2
```

定義可能なシンボルは4000個です。
シンボル1個あたりの記憶域のサイズは、約16バイト必要です。

/o　実行可能プログラムのファイル名の指定

書　式

／o<ファイル名>

機　能

実行可能プログラムのファイル名を指定します。

解　説

リンク時に出力する実行可能プログラムのファイル名を指定します。
指定した実行可能プログラムのファイル名に拡張子が付いていれば、その拡張子となり、拡張子がなければｘとなります。
このスイッチの指定がなければ、実行可能プログラムのファイルは１番目のオブジェクトファイル名の拡張子ｏをｘに置き換えたものとなります。

例

／oスイッチの使用例を以下に示します。

```
LK  /o ldmdl object1 object2
```

実行可能プログラムのファイル名はldmdl. xとなります。

/p　マップファイルの出力

書　式　／p＜ファイル名＞

機　能　マップファイルを出力します。

解　説　実行可能プログラム内のシンボルのオフセット値、セグメントタイプ、シンボル名をファイルに出力します。
マップファイル名は実行可能ファイルの拡張子をmapで置き換えたものとなります。

例　／pスイッチの使用例を以下に示します。

```
LK  /p sample.o ↵
```

と入力すると、sample.mapというファイル名のマップファイルが出力されます。
マップファイルの内容を以下に示します。

```
ADDRESS   TYPE  SYMBOL NAME
00001E36  text  getc         ┐
   .       .      .          ├─TEXT セグメントのシンボル
   .       .      .          │
00001FD0  text  read         ┘
000024AE  data  cellmax      ┐
   .       .      .          ├─DATA セグメントのシンボル
   .       .      .          │
000024BA  data  temppath     ┘
00003C5A  bss   filel        ┐
   .       .      .          ├─BSS セグメントのシンボル
   .       .      .          │
00004A84  bss   stacktop     ┘
   │       │      │
   │       │      └───────────シンボル名
   │       └──────────────────セグメントタイプ
   └──────────────────────────プログラムの先頭からのオフセット
```

/t テンポラリファイルのパス指定

書　式　/t<パス名>

機　能　テンポラリファイルが作られるパスを指定します。

解　説　リンク時に作成されるテンポラリファイルが作られるパスを指定します。
なおテンポラリファイルはリンカの処理終了後に自動的に消去されます。
本スイッチは、環境変数 temp に設定されたディレクトリやカレントディレクトリにテンポラリファイルが作成できない場合（ファイル領域不足によって）指定することができます。

例　/t スイッチの使用例を以下に示します。

```
LK  /t b:temppath sample ↵
```

テンポラリファイルはドライブ b:、パス名 temppath でアクセスされます。

/V バーボーズモード

書式 ／v

機能 バーボーズモードを指定します。

解説 バーボーズモードでは、リンクに関する情報が順次画面に表示されます。

例 ／v スイッチの使用法と、表示される内容を以下に示します。

```
A>LK /V test
X68k Linker v2.00 Copyright 1987,88,89,90 SHARP/Hudson
test.o
          text      00000000-000006B3(000006B3)
          data      000006B4-000006C2(0000000E)
          bss       000006C2-00002C8A(000025C8)
```

└オブジェクトファイル名　　　└機械語サイズ

/X　シンボルテーブルの出力禁止

書　式　／x

機　能　実行可能ファイルにシンボルテーブル及びソースコードデバッグのための拡張シンボル情報を出力しません。

解　説　このスイッチを指定すると、オブジェクトファイル内のシンボルテーブル及び拡張シンボル情報を実行可能ファイルに出力しません。
そのため作成した実行可能ファイルのサイズは小さくなりますが、シンボルを利用したデバッグができなくなるので、十分にデバッグしたプログラムに対してのみ、このスイッチを指定してください。
このスイッチを指定しない場合は、シンボルテーブル及び拡張シンボル情報が出力され、シンボリックデバッグならびにソースコードデバッグが可能となります。

例　／x スイッチの使用例を以下に示します。

```
LK  /x sample ↵
```

5.5 LKエラーメッセージ一覧

エラーメッセージ	意　味
Illeagl option<指定オプション>	オプションスイッチが不正です。
Abort：file open error<ファイルネーム>	ファイルがオープンできません。
Abort：destination file open error <ファイルネーム>	オブジェクトファイルがオープンできません。
Abort：No enough space	インダイレクトファイル解析中にメモリが足りなくなりました。
Abort：Internal file open error	テンポラリファイルがオープンできません。
Abort：Indirect mode error	インダイレクトファイル指定が複数回指定されています。
Abort：Indirect file not found	インダイレクトファイルが見つかりません。
Abort：diskfull	ディスクの空き領域がありません。
Abort：System error	オブジェクトファイルを解析中に不正なオブジェクトコードを発見しました。
Abort：Illegal file error	オブジェクトファイルが不正です。
Abort：mapfile error	マップファイルが作成できません。
Abort：file read error	ファイルが読み込めません。
Abort：file write error	ディスクに書き込めません。
Abort：not enough memory	メモリが足りません。
Abort：Temporary file open error	テンポラリファイルがオープンできません。

上記のエラーが発生した場合は処理を中断します。

5.5 LKエラーメッセージ一覧

エラーメッセージ	意　味
＜シンボルネーム＞Duplicate definition in ＜ファイルネーム＞	外部定義シンボルが重複しています。
Relative addresing overflow in ＜ファイルネーム＞to＜シンボルネーム＞	相対アドレスの相対値が最大値を超えました。
Over flow error in ＜ファイルネーム＞	外部参照を含む式中でオーバーフローしました。
Division by zero in＜ファイルネーム＞	外部参照を含む式中で0除算しました。
Illegal Relocation error ＜シンボルネーム＞	リロケーション情報が不正です。
Illegal exprssion in＜ファイルネーム＞	外部参照を含む式が不正です。
Illegal control command	オブジェクトコード中のコントロールコマンドの部分が不正です。
Undefined symbol(s)	外部参照シンボルが定義されていません。
Relative address in odd address	リロケーションすべきアドレスが奇数アドレスになりました。

第 6 章

デバッガ

使用書式と起動方法
デバッガのスイッチ
コマンドの書式
デバッガのコマンド
DB エラーメッセージ一覧

デバッガ XDebugger は、MPU68000 用シンボリックデバッガであり、ソースプログラムのイメージでデバッグ、具体的にはソースプログラムの変数名やアドレスなどを指定してデバッグを行うことができます。
XDebugger のおもな機能は、次の通りです。

- ● 1 行アセンブル、逆アセンブル機能
- ●ブレークポイント操作
- ●メモリ操作
- ●レジスタ操作
- ●トレース機能
- ●コマンド操作
- ●システム変数操作
- ●計算機能
- ●ファイルアクセス
- ●メモリチェックポイント操作
- ● DOS コマンド実行

6.1 使用書式と起動方法

起動前に確認すること

- デバッガが使用するファイルがディスク上、および処理を行うパス上にありますか?
- パスの中にファイルやDBがテンポラリファイルを作成するのに十分な空き領域がありますか?

デバッガの起動方法と書式

コマンドラインで次のような書式で入力します。

```
DB ［＜スイッチ＞］ ［＜ファイル名＞］ ［＜コマンドライン＞］ ↵
```

6.2 デバッガのスイッチ

デバッガのスイッチには、次のものがあります。

／r　デバッガの入出力をCONからAUXに切り換えてからデバッガを実行します。

コンソールとの入出力を行うプログラムをデバッグする場合、デバッガの表示とデバッグ中のプログラムの表示が重なってしまい非常に見にくくなることがあります。
また、特にグラフィックやスプライトを使用するプログラムでは、テキスト画面を表示しないように設定されることがあるので、そのときデバッガの表示はまったくコンソールに現れません。
このように、デバッガの入出力装置がコンソールでは都合が悪いとき、補助入出力（AUX）をコンソール（CON）の代わりとすることができます。
このスイッチを利用する場合、RS-232Cインターフェースを通じて他の端末に接続する必要があります。
また、X68000とそれに接続する端末の間で転送速度や通信手順を同じにしなければなりません。
X68000と端末の間で正常に通信が行える状態で、X68000のコンソールより／rスイッチを付けてデバッガを実行すると、端末にデバッガのプロンプトが表示され、コマンドが入力可能になります。
以後、デバッガからの出力データはすべて端末に対して出力され、また、端末から入力したコマンドしかデバッガは受け付けません。
デバッガを終了するか、デバッガ上でVコマンドを実行すると、入出力装置がコンソールに戻ります。

／c<メモリサイズ>　OSコマンド実行時のための空きメモリのサイズを指定する

デバッガの!コマンドによりOSのコマンドを実行するために使用するメモリサイズをキロバイト単位で指定します。
省略すると／c128が指定されたことになり、128キロバイトが確保されます。

6.3 コマンドの書式

デバッガのコマンドの書式は次の通りです。

　＜コマンド名＞＜パラメータ＞

＜コマンド名＞は、次ページに記載されている1文字、または2文字のアルファベットによるコマンド名です。
＜パラメータ＞は、コマンドが使用する値、アドレスを表す数値、シンボル、式などです。
＜コマンド名＞や＜パラメータ＞に使用できるシンボルの一覧を次に示します。

演算子

＋　加算
−　減算
＊　乗算
／　除算
＆　論理積
｜　論理和
！　論理否定
＾　排他的論理和
％　剰余

数

16進　数値をそのまま入力します。
10進　数値の先頭に¥を指定します。
2進　数値の先頭に_を指定します。
シンボル　(システム変数、レジスタなど)
　シンボルの先頭に . を指定します。

サイズ

S　バイト
W　ワード
L　ロングワード

6.4 デバッガのコマンド

デバッガのコマンドには次のものがあります。

コマンド	機　能
A ［＜address＞］	アセンブル
AN ［＜address＞］	アセンブル(ニーモック表示なし)
B	ブレークポイントの表示
B ［＜bp＞］ ＜address＞ ［＜count＞］	ブレークポイントの設定
BC＜bp＞	ブレークポイントの削除
BD＜bp＞	ブレークポイントの無効化
BE＜bp＞	ブレークポイントの有効化
BR	ブレークカウントの初期化
C＜string＞	コマンドラインの設定
D ［＜size＞］ ［＜range＞］	メモリ内容のダンプ
F ［＜size＞］ ＜range＞＜data＞	フィルメモリ
G ［＝＜address＞］ ［＜address＞］	デバッグ中のプログラムの実行
H	オンラインヘルプメッセージ
HC	トレース履歴の消去
HI	トレース履歴の表示
L ［＜range＞］	アセンブリリスト表示
ME ［＜size＞］ ［＜address＞］ ［＜data＞］	メモリ内容の編集
MEN ［＜size＞］［＜address＞］	メモリ内容の編集（表示なし）
MM＜range＞＜address＞	メモリ内容の移動
MS ［＜size＞］ ＜range＞＜data＞	メモリ内容の検索
N	メモリチェックポイントの表示
N ［＜size＞］ ［＜cp＞］ ［＜address＞］ ［＜cd＞］ ［＜data＞］	メモリチェックポイントの設定
NC ［＜cp＞］	メモリチェックポイントの消去

コマンド	機　能
ND [<cp>]	メモリチェックポイントの無効
NE [<cp>]	メモリチェックポイントの有効
O	画面表示幅の変更
P	システムステータスの表示
PS	シンボルテーブルの表示
PS<symbol>	シンボルの検索表示
Q	デバッガの終了
R<filename> [,<addrerss>]	ファイルの読み込み
R@<address><drive><record><count>	ディスクの物理リード
S [<count>]	ステップ実行
T [=<address>] [<count>]	トレース実行
U [=<address>] [<count>]	表示なしトレース
V	コンソールの切り替え
W<filename>, <range>	ファイルの書き込み
W@<address><drive><record><count>	ディスクの物理ライト
X	レジスタ内容の表示
X<reg>	レジスタ内容の変更
Y／N	ポーズ
Z	システム変数の表示
Z<num>=<exp>	システム変数の設定
?<exp>	16 進表示
??<exp>	10 進表示
¥	コマンドラインの繰り返し
>filename	出力リダイレクト
>>filename	出力リダイレクト（アペンド）
>@filename	入力コマンドをファイルに保存
<filename	入力リダイレクト
! [<os_command>]	OS コマンドの実行

Assemble

書　　式　A［<アドレス>］
AN［<アドレス>］

機　　能　MPU68000 ニーモニックの 1 行アセンブル

解　　説　MPU68000 のニーモニックをアセンブルしてメモリに設定します。
<アドレス>はアセンブル開始アドレスを指定します。
.PC のようにレジスタ名を指定するとそのレジスタの内容が採用されます。
なお、他のコマンドでも、レジスタ名によるアドレスの指定は可能です。
また、シンボル名でアドレスを指定する場合、ピリオド(.)の後にシンボル名を指定してください。
デバッガのコマンド待ちに戻るときはピリオド(.)を入力してリターンキー↵を押してください。
AN コマンドの機能は上記 A コマンドと同様ですが、A コマンドでは、そのときメモリに格納されているコードのニーモニックが表示されるのに対し、AN コマンドではそれが表示されません。
また、リターンキー↵のみでデバッガのコマンド待ちに戻ります。

例

```
-A ↵
-A 654A0 ↵
-A. PC ↵
```

なおDOSコールニーモニックとFEファンクションコールのニーモニックとして以下のものが用意されており、通常のニーモニックと同様に表記ができます。詳細は別冊のプログラマーズマニュアルを参照してください。

● DOSコールニーモニック

_EXIT	_GETCHAR	_PUTCHAR	_COMINP
_COMOUT	_PRNOUT	_INPOUT	_INKEY
_GETC	_PRINT	_GETS	_KEYSNS
_KFLUSH	_FFLUSH	_CHGDRV	_CHDRV
_DRVCTRL	_CONSNS	_PRNSNS	_CINSNS
_COUTSNS	_FATCHK	_CURDRV	_GETSS
_FGETC	_FGETS	_FPUTC	_FPUTS
_ALLCLOSE	_SUPER	_FNCKEY	_KNJCTRL
_CONCTRL	_KEYCTRL	_INTVCS	_PSPSET
_GETTIM2	_SETTIM2	_NAMESTS	_GETDATE
_SETDATE	_GETTIME	_SETTIME	_VERIFY
_DUP0	_VERNUM	_KEEPPR	_GETDPB
_BREAKCK	_DRVXCHG	_INTVCG	_DSKFRE
_NAMECK	_MKDIR	_RMDIR	_CHDIR
_CREAT	_OPEN	_CLOSE	_READ
_WRITE	_DELETE	_SEEK	_CHMOD
_IOCTRL	_DUP	_DUP2	_CURDIR
_MALLOC	_MFREE	_SETBLOCK	_EXEC
_EXIT2	_WAIT	_FILES	_NFILES
_SETPDB	_GETPDB	_SETENV	_GETENV
_VERIFYG	_COMMON	_RENAME	_FILEDATE
_MALLOC2	_MAKETMP	_NEWFILE	_LOCK
_ASSIGN	_S_MALLOC	_S_MFREE	_S_PROCESS
_EXITVC	_CTRLVC	_ERRJVC	_DISKRED
_DISKWRT	_INDOSFLG	_SUPER_JSR	_BUS_ERR
_OPEN_PR	_KILL_PR	_GET_PR	_SUSPEND_PR
_SLEEP_PR	_SEND_PR	_TIME_PR	_CHANGE_PR

● FE ファンクションコールのニーモニック

__LMUL	__LDIV	__LMOD	__UMUL
__UDIV	__UMOD	__IMUL	__IDIV
__RANDOMIZE	__SRAND	__RAND	__STOL
__LTOS	__STOH	__HTOS	__STOO
__OTOS	__STOB	__BTOS	__IUSING
__LTOD	__DTOL	__LTOF	__FTOL
__FTOD	__DTOF	__VAL	__USING
__STOD	__DTOS	__ECVT	__FCVT
__GCVT	__DTST	__DCMP	__DNEG
__DADD	__DSUB	__DMUL	__DDIV
__DMOD	__DABS	__DCEIL	__DFIX
__DFLOOR	__DFRAC	__DSGN	__SIN
__COS	__TAN	__ATAN	__LOG
__EXP	__SQR	__PI	__NPI
__POWER	__RND	__DFREXP	__DLDEXP
__DADDONE	__DSUBONE	__DDIVTWO	__DIEECNV
__IEEDCNV	__FVAL	__FUSING	__STOF
__FTOS	__FECVT	__FFCVT	__FGCVT
__FTST	__FCMP	__FNEG	__FADD
__FSUB	__FMUL	__FDIV	__FMOD
__FABS	__FCEIL	__FFIX	__FFLOOR
__FFRAC	__FSGN	__FSIN	__FCOS
__FTAN	__FATAN	__FLOG	__FEXP
__FSQR	__FPI	__FNPI	__FPOWER
__FRND	__FFREXP	__FLDEXP	__FADDONE
__FSUBONE	__FDIVTWO	__FIEECNV	__IEEFCNV
__CLMUL	__CLDIV	__CLMOD	__CUMUL
__CUDIV	__CUMOD	__CLTOD	__CDTOL
__CLTOF	__CFTOL	__CFTOD	__CDTOF
__CDCMP	__CDADD	__CDSUB	__CDMUL
__CDDIV	__CDMOD	__CFCMP	__CFADD
__CFSUB	__CFMUL	__CFDIV	__CFMOD
__CDTST	__CFTST	__CDINC	__CFINC
__CDDEC	__CFDEC	__FEVARG	__FEVECS

Display Breakpoint

書　式　B

機　能　ブレークポイントの表示

解　説　B (Set Breakpoint) コマンドで作成した全ブレークポイントの現在の情報を表示します。

このとき表示される情報は以下の通りです。

①ブレークポイント番号
②有効／無効の別
③ブレークポイントのアドレス
④ブレークポイントの通過回数
⑤回数の設定値

ブレークポイントが設定されていないと"no break pointer"と表示されます。

例　Bコマンドの使用例を以下に示します。

```
-B ↵
no break pointer
-■

-B 654A0 5 ↵
-■

-B ↵
0 e 000654A0(0000;0005)
-■
```

Set Breakpoint

書　　式　B［＜ブレークポイント番号＞］［＜アドレス＞］［＜回数＞］

機　　能　ブレークポイントの設定

解　　説　B (Set Breakpoint) コマンドは、入力されたアドレスにブレークポイントを設定します。

プログラム実行時にブレークポイントに達すると、プログラムは停止し、レジスタとフラグの現在値がすべて表示されます。

このブレークポイントは、G コマンドで作成するブレークポイントと違って、Clear Breakpoint コマンドを使用しないかぎり、削除されません。

＜ブレークポイント番号＞には設定するブレークポイントの番号を指定します。

ブレークポイントは 10 個まで設定でき、ブレークポイント番号は 0～9 です。

'B' とブレークポイント番号の間に空白を入れてはいけません。

省略すると、現在使用されていないブレークポイント番号の中で最も小さい値の番号を割り当てます。

＜アドレス＞は、有効な命令のアドレス(すなわち命令コードの最初のバイト)です。

奇数アドレスは指定できません。

＜回数＞は、そのブレークポイントを設定したアドレスを何回通過（実行）されたら停止するか、その回数を指定します。

省略すると 1 となります。

例　B コマンドの使用例を以下に示します。

```
-B ⏎ ─────────────── ブレークポイント作成
no break pointer ──── ブレークポイントなし
-■
```

↓

```
-B 654A0 5 ⏎ ──────── ブレークポイント作成
-■ ─┬─── ┬
    │    └──────────── 回数の設定
    └───────────────── ブレークポイントのアドレス
```

↓

```
-B⏎
0 e 000654A0(0000;0005)
- ■
```

- -B⏎ ―― ブレークポイントの表示
- 0 e 000654A0(0000;0005) ―― ブレークポイントあり
- 0005 ―― 回数の設定値
- 0000 ―― 通過回数
- 000654A0 ―― ブレークポイントのアドレス
- e ―― 有効

Clear Breakpoint

書　　式　BC＜ブレークポイント番号＞［＜ブレークポイント番号＞＜ブレークポイント番号＞……］

機　　能　ブレークポイントの削除

解　　説　BC コマンドは、ブレークポイントを削除します。
＜ブレークポイント番号＞には削除するブレークポイント番号を指定します。
ブレークポイント番号は、複数個指定することができます。
また、アスタリスク（＊）を指定すると、全ブレークポイントが削除されます。

例　BC コマンドの使用例を以下に示します。

```
-B ↵ ──────────────────── ブレークポイントの表示
0 e 000654A0(0000;0005)
1 d 00066000(000A;0007)
2 e 00070000(0000;0001)
-■
```

↓

```
-BC1 ↵ ──────────────────── ブレークポイント1を削除
-■
```

↓

```
-B ↵ ──────────────────── ブレークポイントの表示
0 e 000654A0(0000;0005)
2 e 00070000(0000;0001)
-■
```

Disable Breakpoint

書　　式　BD＜ブレークポイント番号＞［＜ブレークポイント番号＞＜ブレークポイント番号＞……］

機　　能　ブレークポイントの一時的な無効化

解　　説　BDコマンドは、ブレークポイントを一時的に無効とします。このとき、ブレークポイントは削除されるわけではないので、Enable Breakpointコマンドによりいつでも有効にできます。
＜ブレークポイント番号＞には無効とするブレークポイント番号を指定します。ブレークポイント番号は、複数個指定することができます。
また、アスタリスク（＊）を指定すると、全ブレークポイントが無効となります。

例　BDコマンドの使用例を以下に示します。

```
-B⏎ ―――――――――――――――――――― ブレークポイントの表示
0 e 000654A0(0000;0005)
1 e 00066000(000A;0007)
2 e 00070000(0000;0001)
-■
```

↓

```
-BD*⏎ ―――――――――――――――――――― ブレークポイントをすべて無効とする
-■
```

↓

```
-B⏎ ―――――――――――――――――――― ブレークポイントの表示
0 d 000654A0(0000;0005)
1 d 00066000(000A;0007)
2 d 00070000(0000;0001)
-■
```

Enable Breakpoint

書　　式　BE＜ブレークポイント番号＞［＜ブレークポイント番号＞＜ブレークポイント番号＞……］

機　　能　ブレークポイントの有効化

解　　説　BEコマンドは、BDコマンドにより一時的に無効となったブレークポイントを有効にします。
＜ブレークポイント番号＞には有効とするブレークポイント番号を指定します。
ブレークポイント番号は、複数個指定することができます。
また、アスタリスク（*）を指定るすと、全ブレークポイントが有効となります。

例　BEコマンドの使用例を以下に示します。

```
-B⏎ ──────────────────────────── ブレークポイントの表示
0 e 000654A0(0000;0005)
1 e 00066000(000A;0007)
2 d 00070000(0000;0001)
-■
```

↓

```
-BE2⏎ ──────────────────────────── ブレークポイント2の有効化
-■
```

↓

```
-B⏎ ──────────────────────────── ブレークポイントの表示
0 e 000654A0(0000;0005)
1 e 00066000(000A;0007)
2 e 00070000(0000;0001)
-■
```

Break count Reset

書　式　BR

機　能　全ブレークポイントの通過回数クリア

解　説　BRコマンドを実行すると、それまでに通過した全ブレークポイントの通過回数をクリアします。

例　BRコマンドの使用例を以下に示します。

```
-B ⏎ ──────────────────── ブレークポイントの表示
0 e 000654A0(0000;0005)
1 e 00066000(000A;0007)
2 e 00070000(0000;0001)
-■
```

↓

```
-BR ⏎ ──────────────────── ブレークポイントの通過回数クリア
-■
```

↓

```
-B ⏎ ──────────────────── ブレークポイントの表示
0 e 000654A0(0000;0005)
1 e 00066000(0000;0007)
2 e 00070000(0000;0001)
-■
```

Command line set

書　　式　C ［＜文字列＞］

機　　能　デバックの中のプログラムのコマンドラインに＜文字列＞をセットします。

解　　説　デバッグ対象となるプログラムに与えるパラメータに＜文字列＞をセットします。

例　Cコマンドの使用例を以下に示します。

```
-C ABCDEF ⏎
```

Dump memory

書　式　D［<サイズ>］［<レンジ>］

機　能　メモリ内容のダンプ

解　説　メモリ内容を16進と、ASCIIコードでダンプします。
<サイズ>は16進のダンプ単位を指定します(S=バイト、W=ワード、L=ロングワード)。
指定しない場合には、W（ワード）となります。
<レンジ>はダンプの開始アドレスと終了アドレスを指定します。
また、開始アドレスとL長さという指定もできます。
例　DS-50000 L 100

例　Dコマンドの使用例とダンプのフォーマットを以下に示します。

(アドレス$50000～$500FFをバイト単位でダンプ)

```
-DS 50000  500FF ↵
00050000   C3 FC 00 1A D2 BC 00 04 37 A4 22 41 12 29 00 0D
00050010   48 81 B0 41 67 00 00 8E 52 6E FF F4 30 6E FF FE
00050020   22 7C 00 04 C4 72 10 30 98 00 48 80 3D 40 FF F6
00050030   70 01 22 7C 00 04 C4 72 34 6E FF FE D3 CA 12 11
00050040   48 81 C3 FC 00 1A D2 BC 00 04 37 A4 22 41 32 29
00050050   00 06 E3 A0 32 2E FF F6 C3 FC 00 1A D2 BC 00 04
00050060   37 A4 22 41 23 40 00 02 70 01 22 7C 00 04 C4 72
00050070   34 6E FF FE D3 CA 12 11 48 81 C3 FC 00 1A D2 BC
00050080   00 04 37 A4 22 41 32 29 00 08 E3 60 32 2E FF F6
00050090   C3 FC 00 1A D2 BC 00 04 37 A4 22 41 33 40 00 08
000500A0   60 00 00 CA 30 6E FF FE 22 7C 00 04 C4 72 10 30
000500B0   98 00 48 80 C1 FC 00 1A 20 40 22 7C 00 04 37 A4
000500C0   3E B0 98 0A 4E B9 00 03 39 46 20 40 30 28 00 0A
000500D0   30 6E FF FE 22 7C 00 04 C4 72 10 30 98 00 48 80
000500E0   C1 FC 00 1A 20 40 22 7C 00 04 37 A4 3E B0 98 0A
000500F0   4E B9 00 03 39 46 22 40 53 69 00 0A 70 01 22 7C
```

ここにはアスキー文字が表示されます。

(アドレス$60000～$600FFをワード単位でダンプ)

```
-DW 60000  600FF ↵
00060000   C3FC 001A D2BC 0004 37A4 2241 1229 000D
00060010   4881 B041 6700 008E 526E FFF4 306E FFFE
00060020   227C 0004 C472 1030 9800 4880 3D40 FFF6
00060030   7001 227C 0004 C472 346E FFFE D3CA 1211
00060040   4881 C3FC 001A D2BC 0004 37A4 2241 3229
00060050   0006 E3A0 322E FFF6 C3FC 001A D2BC 0004
00060060   37A4 2241 2340 0002 7001 227C 0004 C472
00060070   346E FFFE D3CA 1211 4881 C3FC 001A D2BC
00060080   0004 37A4 2241 3229 0008 E360 322E FFF6
00060090   C3FC 001A D2BC 0004 37A4 2241 3340 0008
000600A0   6000 00CA 306E FFFE 227C 0004 C472 1030
000600B0   9800 4880 C1FC 001A 2040 227C 0004 37A4
000600C0   3EB0 980A 4EB9 0003 3946 2040 3028 000A
000600D0   306E FFFE 227C 0004 C472 1030 9800 4880
000600E0   C1FC 001A 2040 227C 0004 37A4 3EB0 980A
000600F0   4EB9 0003 3946 2240 5369 000A 7001 227C
```

ここにはアスキー文字が表示されます。

（アドレス$60000～$600FF をロングワード単位でダンプ）

```
-DL 60000 600FF⏎
00060000  C3FC001A D2BC0004 37A42241 1229000D
00060010  4881B041 6700008E 526EFFF4 306EFFFE
00060020  227C0004 C4721030 98004880 3D40FFF6
00060030  7001227C 0004C472 346EFFFE D3CA1211
00060040  4881C3FC 001AD2BC 000437A4 22413229
00060050  0006E3A0 322EFFF6 C3FC001A D2BC0004
00060060  37A42241 23400002 7001227C 0004C472
00060070  346EFFFE D3CA1211 4881C3FC 001AD2BC
00060080  000437A4 22413229 0008E360 322EFFF6
00060090  C3FC001A D2BC0004 37A42241 33400008
000600A0  600000CA 306EFFFE 227C0004 C4721030
000600B0  98004880 C1FC001A 2040227C 000437A4
000600C0  3EB0980A 4EB90003 39462040 3028000A
000600D0  306EFFFE 227C0004 C4721030 98004880
000600E0  C1FC001A 2040227C 000437A4 3EB0980A
000600F0  4EB90003 39462240 5369000A 7001227C
```

ここには
アスキー
文字が表
示されます。

なお、＜レンジ＞を省略した場合には、前回のDコマンドで表示された最後のデータの次のアドレスのデータから128バイト分表示されます。
たとえば上記のDコマンド実行後に＜レンジ＞の指定なしでDコマンドを起動すると、アドレス$60100から$6017Fのメモリの内容が表示されます。

```
-D⏎
00060100  0004 C472 346E FFFE D3CA 1211 4881 C3FC
00060110  001A D2BC 0004 37A4 2241 3229 0006 E3A0
00060120  322E FFF6 C3FC 001A D2BC 0004 37A4 2241
00060130  81A9 0002 7001 227C 0004 C472 346E FFFE
00060140  D3CA 1211 4881 C3FC 001A D2BC 0004 37A4
00060150  2241 3229 0008 E360 322E FFF6 C3FC 001A
00060160  D2BC 0004 37A4 2241 8169 0008 526E FFFE
00060170  302E FFFE B079 0003 E14C 6D00 FDE8 302E
```

ここには
アスキー
文字が表
示されます。

Fill memory

書　式　F［＜サイズ＞］＜レンジ＞＜データ＞

機　能　＜レンジ＞内のメモリを＜データ＞で満たします

解　説　Fコマンドは、指定された＜レンジ＞内のアドレスのメモリを＜データ＞の値で満たします。
＜サイズ＞にはデータの指定単位（S＝バイト、W＝ワード、L＝ロングワード）を指定します。
＜サイズ＞の指定を省略するとWとなります。

例　Fコマンドの使用例を以下に示します。

（メモリ内容をダンプ）

```
-DS 60000 6004F↵
00060000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00060010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00060020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00060030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00060040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
-■
```

↓

```
-FS 60000 6002F FF↵
-■
```

↓

（メモリ内容をダンプ）

```
-DS 60000 6004F↵
00060000  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00060010  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00060020  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00060030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00060040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
-■
```

Go

書式 G［=<開始アドレス>］［<ブレークポイントのアドレス>］

機能 デバッグ中のプログラムの実行

解説 Gコマンドは、現在デバッグ中のプログラムを実行します。
<開始アドレス>を指定した場合は、指定した開始アドレスから、開始アドレスを省略した場合はプログラムカウンタの現在値が示すアドレスから実行が開始されます。
実行は、プログラムが終了するまで、または<ブレークポイントのアドレス>で指定したブレークポイントで停止するまで続きます。
なお、ブレークポイントはBコマンドの他に、Gコマンドでも指定することができます。
また、Gコマンドで指定したブレークポイントは指定時のみ有効であり、再度停止させたい場合には次のGコマンドで再度指定する必要があります。
プログラムの実行中にブレークポイントに達すると、レジスタとフラグの現在値、および次に実行する命令が表示されます。
表示形式は、Xコマンドの場合と同じです。

例 Gコマンドの使用例を以下に示します。

```
-G=5432A 5443C ↵
```

アドレス$5443Cにブレークポイントを設定した後、アドレス$5432Aよりプログラムを実行します。

Help

書　式　H

機　能　デバッガコマンドの表示

解　説　Hコマンドは、デバッガコマンドのリストを表示します。

例　Hコマンドの使用例を以下に示します

```
-H ⏎
A[address]                          :アセンブル
AN[address]                         :アセンブル(ニーモニック表示なし)
B                                   :ブレークポイントの表示
B[bp] address [count]               :ブレークポイントの設定
BC bp                               :ブレークポイントの削除
BD bp                               :ブレークポイントの無効化
BE bp                               :ブレークポイントの有効化
BR                                  :ブレークカウントの初期化
C string                            :コマンドラインの設定
D[size][range]                      :メモリ内容のダンプ
F[size]range data                   :フィルメモリ
G[=address] [address]               :デバッグ中のプログラムの実行
H                                   :オンラインヘルプメッセージ
HC                                  :トレース履歴の消去
HI                                  :トレース履歴の表示
L[range]                            :アセンブリリスト表示
ME[size][address] [data]            :メモリ内容の編集
MEN[size][address]                  :メモリ内容の編集(表示なし)
MM range address                    :メモリ内容の移動
MS[size]range data                  :メモリ内容の検索
N                                   :メモリチェックポイントの表示
N[size][cp] address [cd][data]      :メモリチェックポイントの設定
NC cp                               :メモリチェックポイントの消去
ND cp                               :メモリチェックポイントの無効
NE cp                               :メモリチェックポイントの有効
O                                   :画面表示幅の変更
size            s(byte) w(word) l(long)
bp              0-9
cd(condition)   0:<>,1:=
cp              0-9
-- More -- (y/n) ?
P                                   :システムステータスの表示
PS                                  :シンボルテーブルの表示
PS symbol                           :シンボルの検索表示
Q                                   :デバッガの終了
R filename[,address]                :ファイルの読み込み
R@ address drive record count       :ディスクの物理リード
S[count]                            :ステップ実行
T[=address] [count]                 :トレース実行
U[=address] [count]                 :表示なしトレース
V                                   :コンソールの切り替え
W filename,range                    :ファイルの書き込み
W@ address drive record count       :ディスクの物理ライト
X                                   :レジスタ内容の表示
X reg                               :レジスタ内容の変更
Y/N                                 :ポーズ
Z                                   :システム変数の表示
Z num=exp                           :システム変数の設定
? exp                               :16進表示
?? exp                              :10進表示
¥                                   :コマンドラインの繰り返し
> filename                          :出力リダイレクト
```

```
>> filename                          : 出力リダイレクト（アペンド）
>@ filename                          : 入力コマンドをファイルに保存
< filename                           : 入力リダイレクト
![os_command]                        : OSコマンドの実行
symbol          シンボル名の先頭に . (period)
drive           0:current,1:A,2:B, .....
reg             d0-d7 a0-a7 ssp usp sr ccr pc
operators       + - * / & (and) | (or) ! (not) % (residue) ^ (exor)
number          ??(hex.) .??(symbol) ¥??(dec.) _??(bin.)
-■
```

History Clear

書　式　HC

機　能　トレース履歴の消去

解　説　トレース履歴を消去します。

例　HCコマンドの使用例を以下に示します。

```
-HI ↵
Trace history from older
00000000
000654A0
000654A6
000654AE
00065406
0006540C
00065412
00065418

-HC ↵
-HI ↵
Trace history from older
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000

- ■
```

History trace

書　式　HI

機　能　トレース履歴の表示

解　説　このコマンドを実行するまでにトレースした機械語のアドレスを、8個まで表示します。

例　HIコマンドの使用例を以下に示します。

```
-HI ↵
Trace history from older
00000000
000654A0
000654A6
000654AE
00065406
0006540C
00065412
00065418
```

List

書　式　L［＜レンジ＞］

機　能　メモリ内容の逆アセンブルリスト

解　説　Lコマンドは、指定された＜レンジ＞内の、デバッグ中のプログラムのソースコードを表示します。

＜レンジ＞には、逆アセンブルの開始アドレスと終了アドレスを指定します。

＜レンジ＞の指定が行われない場合、現在の逆アセンブルアドレスの命令を表示します。

現在の逆アセンブルアドレスとは、前回のLコマンド実行時に表示された最終バイト(行)の次のバイト(行)のことです。

なお、MPU68000ニーモニックコードの表示のみが可能です。

例　Lコマンドの使用例を以下に示します。

```
-L 60400 6040F↵
00060400          movea.l D6,A0
00060402          move.l  $20(A1,D2),-(A6)
00060406          sub.w   (A6),D0
00060408          suba.l  (A2),A2
0006040A          or.w    D7,(A7)
0006040C          movep.w $5348(A2),D6
-■
```

Memory Edit

書　式　ME［＜サイズ＞］［＜アドレス＞］［＜データ＞...］
MEN［＜サイズ＞］［＜アドレス＞］

機　能　メモリ内容の編集

解　説　メモリ内容をエディットします。
＜サイズ＞はエディット単位を指定します。
(S=バイト、W=ワード、L=ロングワード)
＜アドレス＞はエディット開始アドレスを指定します。
なお、＜データ＞には 'ABCD' のように文字列を指定することもできますが、その場合には、どのサイズを指定してもバイト単位となります。
＜データ＞を指定しない場合には、エディット状態になります。
抜けるときには、ピリオドを入力して[↵]を押してください。

MEN コマンドは ME コマンドと機能は同様ですが、ME コマンドではエディット時現在のメモリの内容が表示されますが、MEN コマンドでは表示されません。

MEP［＜アドレス＞］［＜データ＞］
MENP［＜アドレス＞］

上記のように、サイズにPを指定した場合には、バイトアクセスでアドレスが2増加します。
おもに、I/O ポート用に使います。

例　ME コマンドおよび MEN コマンドの使用例を以下に示します。
データを指定しないと現在メモリに設定されている内容が表示されデータの入力を促します。

```
-MES 60500↵
00060500        FF :00↵
00060501        FF :00↵
00060502        FF :.↵
-■

-MEW 60502↵
00060502        FFFF :0000↵
00060504        FFFF :.↵
-■

-MEL 60504↵
00060504        FFFFFFFF :00000000↵
00060508        FFFFFFFF :.↵
-■
```

アドレスを指定しないと、前回のME(MEN)コマンドでエディット済みのデータの次のアドレスのデータよりエディットします。
すなわち、上記の例でいうと、上記のコマンド実行後に、アドレス指定なしでME(MEN)コマンドを再度起動すると、アドレス$60508のデータよりエディットします。

```
-ME↵
00060508        FFFF :■
```

また、上記コマンドのようにサイズの指定を省略すると、ワード単位でエディットが行われます。

MEコマンドでデータを指定したとき、またはMENコマンドを実行したときには、現在のメモリ内容の表示は行われません。

```
-MEL 60508 00000000↵
-■

-MEN↵
0006050C        :0000↵
0006050E        :.↵
-■

-MENW 6050E↵
0006050E        :■
```

Memory Move

書　式　MM <レンジ> <アドレス>

機　能　メモリ内容の移動

解　説　MM コマンドは、<レンジ>の指定するメモリブロックを、<アドレス>で指定した位置へ移動します。
<レンジ>で指定した移動元アドレスのメモリの内容は、本コマンド実行後もそのまま残っています。
転送元ブロックと転送先ブロックが一部重なっている場合にも正しく転送されます。

例　MM コマンドの使用例を以下に示します。

```
00060400   2046 2D31 2020 9056 9D52 8F57 0D0A 5348
00060410   0000 0000 0000 0000 0000 0000 0000 0000
00060420   0000 0000 0000 0000 0000 0000 0000 0000
00060430   0000 0000 0000 0000 0000 0000 0000 0000
00060440   0000 0000 0000 0000 0000 0000 0000 0000
00060450   0000 0000 0000 0000 0000 0000 0000 0000
```

↓

```
-MM 60400 6040F 60430⏎
```

アドレス$60400～$6040F のデータをアドレス$60430 以降に移動

↓

```
00060400   2046 2D31 2020 9056 9D52 8F57 0D0A 5348
00060410   0000 0000 0000 0000 0000 0000 0000 0000
00060420   0000 0000 0000 0000 0000 0000 0000 0000
00060430   2046 2D31 2020 9056 9D52 8F57 0D0A 5348
00060440   0000 0000 0000 0000 0000 0000 0000 0000
00060450   0000 0000 0000 0000 0000 0000 0000 0000
```

(MM コマンド実行後)

Memory Search

書　式　MS［<サイズ>］<レンジ><データ>

機　能　メモリ内容の検索

解　説　MSコマンドは、<レンジ>で指定したメモリ内を検索し、<データ>で指定した値を探します。
<サイズ>はS、W、L(S＝バイト、W＝ワード、L＝ロングワード)のどれかで、省略した場合にはWとなります。
<データ>には複数のデータを指定できます。
また'ABC'のように文字列を指定することもできます。
ただし、この場合サイズ指定は自動的にSとなります。

例　MSコマンドの使用例を以下に示します。

```
-MS 64000 64100  'equ'↵
```

と入力すると、'equ'というデータが存在する領域の先頭アドレスがすべて表示されます。

```
00064012 00064050 0006408E 000640CC
```

Display Memory Check Point

書　式　N

機　能　メモリチェックポイントの表示

解　説　N (Set Memory Check Point) コマンドで設定した全てのメモリチェックポイントの現在の情報を表示します。
この時表示される情報を以下に示します。

①メモリチェックポイント番号
②有効／無効の別
③メモリチェックポイントのアドレス
④メモリチェックポイントのサイズ
⑤チェック条件
⑥比較値
⑦メモリチェックポイントの現在の内容

メモリチェックポイントが1つも設定されていないと、"no memory check pointer"と表示されます。

例　N コマンドの使用例を以下に示します。

```
-N⏎
0 e 000024A0(w 0 00006020)        ;000024A0(6020)
1 d 00054321(s 1 00000072)        ;00054321(72)
2 e 00068000(l 0 60FFFFFE)        ;00068000(2F3C0007)
-■
```

Set Memory Check Point

書　式	N［＜サイズ＞］［＜番号＞］＜アドレス＞［＜条件＞］［＜比較値＞］
機　能	メモリチェックポイントの設定

解　説

N (Set Memory check point) コマンドは、指定されたアドレスのメモリ内容が変化したときにプログラムを止めるために、メモリチェックポイントを設定するコマンドです。

プログラムのトレースを実行中に、このコマンドで指定されたチェックポイントのメモリ内容が、指定された条件に合った場合、プログラムは停止します。

＜サイズ＞には、メモリチェックポイントのサイズを指定します。

(S＝バイト、W＝ワード、L＝ロング)

省略すると、サイズはワードになります。

＜番号＞には、設定するメモリチェックポイントの番号を指定します。

メモリチェックポイントは10個まで設定でき、メモリチェック番号は0～9です。

'N' とメモリチェック番号の間に空白を入れてはいけません。

省略すると、現在使用されていないメモリチェックポイント番号の中で最も小さい番号を割り当てます。

＜アドレス＞は、チェックしたいメモリ領域の先頭アドレスを指定します。

＜条件＞は、0か1を指定します。

0を指定すると、チェックポイントのメモリ内容が比較値と違う値に変化した時 (Not Equal, ＜＞)、トレースが停止します。

1を指定すると、メモリ内容が比較値と等しい値に変化した時 (Equal,＝＝)、トレースが停止します。

省略すると0が設定されます。

＜比較値＞には、メモリ内容をチェックするときに比較する値を指定します。

省略すると現在のメモリ内容の値がそのまま設定されます。

例

Nコマンドの使用例を以下に示します。

```
-N ↵
no memory check pointer
-N 68000 1↵
-NL8 8100 0 60FFFFFE ↵
-N ↵
0 e 00068000(w 1 00006020)       ;00068000(6020)
8 e 00008100(l 0 60FFFFFE)       ;00008100(2F3C0007)
- ■
```

Clear Memory Check Point

書　式　NC<番号 1>［<番号 2><番号 3>……］

機　能　メモリチェックポイントの削除

解　説　NC コマンドは、指定された番号のメモリチェックポイントを削除します。
<番号>には削除するメモリチェックポイント番号を指定します。
<番号>は複数個指定できます。
また、アスタリスク（＊）を指定すると、全てのメモリチェックポイントを削除します。

例　NC コマンドの使用例を以下に示します。

```
-N ⏎
0 e 00068000(w 1 00006020)       ;00068000(6020)
8 e 00008100(l 0 60FFFFFE)       ;00008100(2F3C0007)
-NC 0 ⏎
-N ⏎
8 e 00008100(l 0 60FFFFFE)       ;00008100(2F3C0007)
-NC * ⏎
-N ⏎
no memory check pointer
- ■
```

Disable Memory Check Point

書　　式　ND＜番号1＞［＜番号2＞＜番号3＞……］

機　　能　メモリチェックポイントの一時的な無効化

解　　説　NDコマンドは、メモリチェックポイントを一時的に無効にします。
削除されるわけではないので、NEコマンドにより、いつでも有効化できます。
＜番号＞には無効化するメモリチェックポイント番号を指定します。＜番号＞は複数個指定できます。
また、アスタリスク（＊）を指定すると、全てのメモリチェックポイントを無効化します。

例　NDコマンドの使用例を以下に示します。

```
-N ↵
0 e 00068000(w 1 00006020)      ;00068000(6020)
8 e 00008100(l 0 60FFFFFE)      ;00008100(2F3C0007)
-ND 0 ↵
-N ↵
0 d 00068000(w 1 00006020)      ;00068000(6020)
8 e 00008100(l 0 60FFFFFE)      ;00008100(2F3C0007)
-ND *↵
-N ↵
0 d 00068000(w 1 00006020)      ;00068000(6020)
8 d 00008100(l 0 60FFFFFE)      ;00008100(2F3C0007)
- ■
```

Enable Memory Check Point

書式　NE＜番号＞［＜番号2＞＜番号3＞……］

機能　メモリチェックポイントの有効化

解説　NEコマンドは、メモリチェックポイントを有効にします。
＜番号＞には有効化するメモリチェックポイント番号を指定します。
＜番号＞は複数個指定できます。
また、アスタリスク（＊）を指定すると、全てのメモリチェックポイントを有効化します。

例　NEコマンドの使用例を以下に示します。

```
-N ⏎
0 d 00068000(w 1 00006020)        ;00068000(6020)
8 d 00008100(l 0 60FFFFFE)        ;00008100(2F3C0007)
-NE 0 ⏎
-N ⏎
0 e 00068000(w 1 00006020)        ;00068000(6020)
8 d 00008100(l 0 60FFFFFE)        ;00008100(2F3C0007)
-NE * ⏎
-N ⏎
0 e 00068000(w 1 00006020)        ;00068000(6020)
8 e 00008100(l 0 60FFFFFE)        ;00008100(2F3C0007)
- ■
```

Change Display Width

書　　式　O

機　　能　画面表示幅を変更する

解　　説　各コマンドを実行したときに画面に表示されるメッセージを、半角文字64桁分の幅に収まるよう変更します。

デバッガ起動直後にOコマンドを実行すると、表示幅が96桁から64桁に切り替わります。

もう1回Oコマンドを実行すると、表示幅が96桁に切り替わります。

このコマンドは、画面モードを64桁モードに切り変えてしまうようなプログラムをデバッグする際に使用します。

例　Oコマンドの使用例を以下に示します。

```
-D 000000 L40⏎
00000000  00FF 0540 01FF 0540 0007 5032 0007 5040      ...@...@..P2..P@
00000010  0007 5052 0007 506C 0007 507C 0007 5090      ..PR..Pl..P|..P.
00000020  0007 50A6 0007 50BE 0007 50C8 0005 A9A4      ..Pｦ..Pｾ..Pﾈ..ｩ､
00000030  0CFF 0540 0DFF 0540 0EFF 0540 0FFF 0540      ...@...@...@...@
-O⏎
-D 000000 L40⏎
00000000  00FF 0540 01FF 0540      ...@...@
00000008  0007 5032 0007 5040      ..P2..P@
00000010  0007 5052 0007 506C      ..PR..Pl
00000018  0007 507C 0007 5090      ..P|..P.
00000020  0007 50A6 0007 50BE      ..Pｦ..Pｾ
00000028  0007 50C8 0005 A9A4      ..Pﾈ..ｩ､
00000030  0CFF 0540 0DFF 0540      ...@...@
00000038  0EFF 0540 0FFF 0540      ...@...@
-■
```

Print status

書　　式　P

機　　能　デバッガおよびデバッグ中のプログラムのアドレス表示

解　　説　デバッガとデバッグ中のプログラムの先頭アドレス、終了アドレス、実行アドレス等を表示します。

例　Pコマンドの使用例を以下に示します。

```
-P ↵
debug  program from $0005E9B0 ――――デバッガの先頭アドレス
user   program from $000671E0 ――――ユーザープログラムの先頭アドレス
               end  $00068BA8 ――――ユーザープログラムの終了アドレス
               exec $000671E0 ――――ユーザープログラムの実行アドレス
symbol   table from $000653B0 ――――シンボルテーブルがない場合は表示されません
```

Print Symbol

書　式	PS
機　能	シンボルテーブルの表示
解　説	シンボルテーブルに登録されているシンボル名の一覧を表示します。

例

```
-PS ↵
$00000000 : ABCD
$00000000 : FUNC_S
$00000000 : MAIN
          .
          .
          .
```

Search Print Symbol

書　　式　PS<シンボル>

機　　能　指定されたシンボルをサーチし表示する。

解　　説　<シンボル>で指定した、シンボル名をシンボルテーブルからサーチして、みつかった場合表示します。
なお<シンボル>には、シンボル名の先頭何文字かを指定し、その文字が含まれるシンボル名をサーチできます。

例　FALSEという名のシンボルをサーチ

```
-PS FALSE ⏎
$00000000 : FALSE
```

"F"で始まる全シンボルをサーチ

```
-PS F ⏎
$00000000 : FALSE
$00000000 : FUNC_S
$00000002 : FLOAT
```

Quit

書　　式　Q

機　　能　デバッガの終了

解　　説　Qコマンドは、デバッガを終了させ、親プロセスに制御を戻します。

例　Qコマンドは引数なしで使用します。

```
-Q ⏎
```

Read file

書式　R ＜ファイル名＞ [,＜アドレス＞]

機能　ファイルを読み込む

解説　＜ファイル名＞で指定したファイルを＜アドレス＞から読み込みます。
＜アドレス＞を指定しない場合、以下のように読み込むファイルの種類により動作が異なります。

● X 形式または R 形式の実行可能ファイル
G コマンド等で実行できるように変換してユーザープログラム領域にロードします。
ユーザープログラム領域は P コマンドを実行した時の"user program from $??????"という表示で知ることができます。

● Z 形式の実行可能ファイル
ファイルのヘッダに格納されているロードアドレスがユーザープログラム領域内であれば、ロードアドレスよりロードします。
この場合そのまま G コマンド等で実行することができます。
ロードアドレスがユーザープログラム領域外であれば、ファイルの内容をそのままユーザープログラム領域にロードします。
この場合、G コマンド等で実行することはできません。

●実行できないファイル
ファイルの内容をそのままユーザープログラム領域にロードします。
もちろん、G コマンド等で実行することはできません。

＜アドレス＞を指定した場合は、指定したファイルの種類に関係なく、ファイルの内容をそのまま加工しないで、指定されたアドレスよりロードします。
この場合、R 形式の実行可能ファイルをロードした時のみ G コマンド等で実行することができます。
OS やデバッガ等が使用しているアドレスを指定してはいけません。

例　R コマンドの使用例を以下に示します。

```
-R sample.x⏎
No symble file
PC=00098B00 USP=00077244 SSP=000067F2 SR=0000 X:0 N:0 Z:0 V:0 C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  00098A00 00098B02 00077BD4 0006F1D0  00098B00 00000000 00000000 00077244
bra.s    $00098B00
- ■
```

Disk Physical Read

書式 R@ <アドレス> <ドライブ番号> <レコード番号> <レコード数>

機能 ディスクの物理的読み込み

解説 R@コマンドは、指定されたパラメータに従ってディスクの内容をメモリに読み込みます。

<アドレス>には、ディスクから読み込むデータを格納するメモリ領域の先頭アドレスを指定します。

<ドライブ番号>には、以下に示すドライブ番号を指定します。

0：カレントドライブ
1：Aドライブ
2：Bドライブ

なお、仮想ドライブや仮想ディレクトリに割り当てられている実ドライブはアクセスできません。

<レコード番号>は読み込む先頭のレコードの番号を16進数で指定します。

<レコード数>は読み込むレコード数を16進数で指定します。

```
-FS A0000 A0FFF 0↵
-R@ A0000 1 0 1↵
-DS A0000 L200↵
000A0000  60 1C 48 75 64 73 6F 6E 20 73 6F 66 74 20 32 2E  ..Hudson soft 2.
000A0010  30 30 04 00 01 02 00 01 00 C0 04 D0 FE 02 4F FA  00.......ﾀ.ﾐ..O.
000A0020  FF E0 43 FA 01 3C 61 00 01 30 24 3C 03 00 00 06  .澆..<a..0$<....
000A0030  20 3C 00 00 00 8E 4E 4F 12 00 E1 41 12 3C 00 70   <...晒O..鰯.<.p
000A0040  43 FA 01 1C 32 81 26 3C 00 00 04 00 43 FA 02 1E  C...2<....C...
000A0050  61 00 00 D4 4A 80 66 00 00 E8 43 FA 02 10 3C 3C  a..ﾔJ  .._鑁...<<
000A0060  00 1F 24 49 47 FA 01 F9 7E 0A 10 1A 80 3C 00 20  ..$IG...‾...   .
000A0070  B0 1B 66 06 51 CF FF F4 60 2A D3 FC 00 00 00 20  -.f.Qﾏ.  *ﾓ....
000A0080  51 CE FF E0 43 FA 01 4C 2F 09 43 FA 00 D6 61 00  Qﾎ.澆..L/.C..ﾖa.
000A0090  00 C8 22 5F 61 00 00 C2 32 3A 00 C4 70 4F 4E 4F  .ﾈ"_a..ﾂ2:.ﾄpONO
000A00A0  70 FE 4E 4F 30 29 00 1A E1 58 55 40 D0 7C 00 0B  p.NO0)..畊U@ﾐ|..
000A00B0  34 00 C4 7C 00 07 52 02 E8 48 64 04 84 7C 01 00  4.ﾄ|..R.鍠d.л..
000A00C0  48 42 34 3C 03 00 14 00 48 42 48 E7 70 00 43 FA  HB4<....HBH輛.C.
000A00D0  01 9C 26 3C 00 00 04 00 61 4C 4C DF 00 0E 43 FA  .<....aLL゜..C.
000A00E0  01 8C 0C 59 48 55 66 60 54 89 0C 99 00 00 68 00  .YHUf.T.h.
000A00F0  66 5E 2F 19 26 19 D6 99 2F 03 2F 19 22 7C 00 00  f^/.&.ﾖ./."|..
000A0100  67 C0 D6 BC 00 00 00 40 61 1C 22 1F 24 1F 22 5F  gﾀﾖｼ...@a.".$."_
000A0110  4A 80 66 2C 41 F9 00 00 68 00 D1 C2 53 81 65 04  J  ,A...h.ﾑﾂS ‘.
000A0120  42 18 60 F8 4E D1 48 E7 78 40 70 46 2F 09 4E 4F  B...NﾑH輾@pF/.NO
000A0130  22 5F 08 00 00 1E 66 02 42 80 4C DF 02 1E 4E 75  "_....f.B ゜..Nu
000A0140  43 FA 00 B4 60 00 FF 42 43 FA 00 CB 60 00 FF 3A  C..ｴ...BC..ﾋ...:
000A0150  43 FA 00 E5 60 00 FF 32 70 21 4E 4F 4E 75 90 70  C..頼..2p!NONu壬
000A0160  1A 00 1B 5B 34 37 6D 1B 5B 31 33 3B 32 36 48 20  ...[47m.[13;26H
000A0170  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
000A0180  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
000A0190  20 20 20 20 20 20 20 20 20 20 20 20 1B 5B 31 34              .[14
000A01A0  3B 32 36 48 20 20 20 20 20 20 20 20 20 20 20 20  ;26H
000A01B0  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
000A01C0  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
000A01D0  20 00 1B 5B 31 34 3B 33 35 48 48 75 6D 61 6E 2E   ..[14;35HHuman.
000A01E0  73 79 73 20 82 AA 20 8C A9 82 C2 82 A9 82 E8 82  sys が 見つかりま
000A01F0  DC 82 B9 82 F1 00 1B 5B 31 34 3B 33 38 48 83 66  ﾜせん..[14;38Hデ
-■
```

Step

書式　S［＜カウント＞］

機能　プログラムのステップ実行

解説　Sコマンドは、プログラムを1ステップづつ実行し、そのつどレジスタとフラグの現在値をすべて表示します。
Tコマンドと違う点は、SコマンドではJSRやBSR命令を1命令として実行し、サブルーチンに分岐しません。
＜カウント＞を指定すると、その数の命令だけステップします。

例　Sコマンドの使用例を以下に示します。

```
-S↵
PC=00070B70 USP=00076D58 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  0005FB00 00076D58 0003ECD8 000362D0  00070B5E 00000000 00000000 00076D58
bsw.w    $00070DE6
-↵
bsw.w    $00070DE6
PC=00070B74 USP=00076D58 SSP=000067F2 SR=0000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  0005FB00 00076D58 0003ECD8 000362D0  00070B5E 00000000 00000000 00076D58
lea      $0100(A0),A5              ;0005FC00(60)
-S↵
PC=00070B78 USP=00076D58 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000001 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  0005FB00 00076D58 0003ECD8 000362D0  00070B5E 0005FC00 00000000 00076D58
move.l   A5,$000768EC              ;000768EC(00000000)
-S↵
```

Trace

書　式　T［=＜アドレス＞］［＜カウント＞］

機　能　プログラムの実行とトレース

解　説　Tコマンドは、プログラムを1命令ずつ実行し、そのつどレジスタとフラグの現在値をすべて表示します。

＜アドレス＞を指定した場合は、そのアドレスからトレースを開始します。

指定しない場合には、プログラムカウンタの示すアドレスからトレースを開始します。

＜アドレス＞を指定する場合は、イコール記号（=）が必要となります。

＜カウント＞を指定すると、その数の命令を実行するまで停止しません。

例　Tコマンドの使用例とトレースの表示形式を以下に示します。

```
-T=99FD2 3↵
PC=00099FD2 USP=00078B18 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000061 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  00099ED0 00099FEA 00079428 00070FB0  00099FD0 00000000 00000000 00078B18
and.b    #$DF,D0
PC=00099FD6 USP=00078B18 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000041 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  00099ED0 00099FEA 00079428 00070FB0  00099FD0 00000000 00000000 00078B18
cmp.b    #$1A,D0
PC=00099FDA USP=00078B18 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000041 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  00099ED0 00099FEA 00079428 00070FB0  00099FD0 00000000 00000000 00078B18
beq.s    $00099FE8
-■
```

Untrace

書　　式　U［=＜アドレス＞］［＜カウント＞］

機　　能　表示なしトレース

解　　説　Uコマンドの機能はTコマンドとほぼ同じですが、プログラムの実行が停止するまでトレースの表示は行いません。
すなわち、Tコマンドのように1命令実行するたびにトレース表示するということはしません。
＜カウント＞を指定して複数の命令をトレースする際、トレース表示が必要ない場合に、このUコマンドを使用します。

例　Uコマンドの使用例とトレースの表示形式を以下に示します。

```
-U=99FD2 3↵
PC=00099FDA USP=00078B18 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000041 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  00099ED0 00099FEA 00079428 00070FB0  00099FD0 00000000 00000000 00078B18
beq.s   $00099FE8
-■
```

Console Change

書　式　V

機　能　コンソール切り替え(CON ⟷ AUX)

解　説　コマンドの入出力装置をCONとAUXの間で切り換えます。
AUXに切り替わると、それ以後デバッガに対する入出力は、RS-232Cポートを介して行われるようになります。
コンソールとの入出力を行うプログラムをデバッグする時に、プログラムとデバッガの表示が重なってしまい非常に見にくくなることがあります。
また、特にグラフィックやスプライトを使用するプログラムでは、テキスト画面を表示しないように設定することがあるので、そのときデバッガの表示はまったくコンソールに現れません。
このように、デバッガとのインターフェースがコンソールでは都合が悪いとき、補助入出力（AUX）をコンソールの代わりにすることができます。
このコマンドを利用する場合、RS-232Cインターフェースを通じて他の端末に接続する必要があります。
また、X68000とそれに接続する端末の間で転送速度や通信手順を同じにしなければなりません。
X68000と端末の間で正常に通信が行える状態で、Vコマンドを実行すると、端末にデバッガのプロンプトが表示され、コマンドが入力可能になります。
以後、デバッガからの出力データはすべて端末に対して出力され、また、端末から入力したコマンドしかデバッガは受け付けません。
元のコンソール入出力に戻る場合は、端末よりVコマンドを実行します。

例　Vコマンドの使用例を以下に示します。

X68000側の表示	→	端末側の表示
-v ↵		-■

Write file

書　式　W <ファイル名>,<レンジ>

機　能　メモリ内容のファイルへの書き出し

解　説　メモリの内容をファイルに書き出します。
<レンジ>で指定したメモリブロックの内容を<ファイル名>で指定したファイルに書き込みます。
メモリ上にロードした実行可能プログラム（X形式またはZ形式）を、このコマンドでファイルに書き出しても、そのファイルは実行できません。
なぜなら、X形式やZ形式の実行可能ファイルには、メモリ上にプログラムをロードするときの再配置情報が付属しており、メモリ上にロードされたときの内容とファイルの内容が違うためです。

例　Wコマンドの使用例を以下に示します。
アドレス$60000～$603FFのメモリの内容をファイル名 'USERFILE' に書き出します。

```
-W USERFILE,60000,603FF↵
-■
```

注：デバッグ中のXファイルをこのコマンドで書き込んではいけません。なぜなら、X形式のファイルにはローディング時にメモリに再配置するための情報が含まれており、実際にメモリ上に置かれた内容とは違っているからです。

Disk Physical Write

書　式　W@ ＜アドレス＞ ＜ドライブ番号＞ ＜レコード番号＞ ＜レコード数＞

機　能　ディスクの物理的書き込み

解　説　W@コマンドは、指定されたパラメータに従ってメモリの内容をディスクに書き込みます。

＜アドレス＞には、ディスクに書き込むデータを格納してある領域の先頭アドレスを指定します。

＜ドライブ番号＞には、以下に示すドライブ番号を指定します。

0：カレントドライブ
1：ドライブA
2：ドライブB

なお、仮想ドライブや仮想ディレクトリに割り当てられている実ドライブはアクセスできません。

＜レコード番号＞は書き込み先頭レコードの番号を16進数で指定します。

＜レコード数＞は書き込むレコード数を16進数で指定します。

例　-W@コマンドの例を示します。

```
-W@　A0000 1 20 10↵
```

アドレス$A0000からのメモリの内容をドライブAのレコード番号$20から$10レコード分書き出します。

Display Register

書　式　X

機　能　全レジスタの内容の表示

解　説　X コマンドは、レジスタとフラグの内容をすべて表示します。

例　X コマンドの使用例と表示形式を以下に示します。

```
 -X⏎ ①            ②             ③              ④        ⑤      ⑥      ⑦      ⑧      ⑨
 PC=0009A000 USP=00078B18 SSP=000067F2 SR=0000 X:0   N:0   Z:0   V:0   C:0
⑩D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
⑪A  00099F00 0009A05C 000700B2 00070FB0  0009A000 00000000 00000000 00078B18
 pea     (A5)             ;00000000(00)
 -■
```

①プログラムカウンタ
②ユーザースタックポインタ
③システムスタックポインタ
④ステータスレジスタ
⑤エクステンドフラグ（ステータスレジスタのビット 4)
⑥ネガティブフラグ　（ステータスレジスタのビット 3)
⑦ゼロフラグ　　　　（ステータスレジスタのビット 2)
⑧オーバフローフラグ（ステータスレジスタのビット 1)
⑨キャリーフラグ　　（ステータスレジスタのビット 0)
⑩データレジスタ　　(d0～d7)
⑪アドレスレジスタ　(a0～a7)

Register Change

書　式　X <レジスタ>

機　能　レジスタの内容の変更

解　説　Xコマンドは、指定されたレジスタの内容を変更します。
指定可能なレジスタ名は以下の通りです。

d0～d7　a0～a7　ssp　usp　sr　ccr　pc

レジスタの内容を変更するときは、コマンドとレジスタ名を入力してください。入力すると、レジスタ名、その現在値、プロンプトが表示されますので、新しい値を指定し、リターンキーを押してください。

例　Xコマンドの使用例を以下に示します。

レジスタの内容を表示

```
-X ⏎
PC=0008A8B6 USP=00089E1E SSP=000757CA SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  00000000 0008A8CA 0008A72E 000838B0  0008A8B0 00000000 00000000 00089E1E
cmp.b   #$1A,D0
-■
```

↓

```
-X A0 ⏎                 レジスタ A0 の変更要求
A0:00000000        ■    レジスタ A0 の現在値を表示し、入力を促します
```

↓

```
A0:00000000      0008A7B0 ⏎     レジスタ A0 に値を設定し、リターンキーを押します
-■
```

↓

レジスタの内容を表示

```
-X ↵
PC=0008A8B6 USP=00089E1E SSP=000757CA SR=8000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  0008A7B0 0008A8CA 0008A72E 000838B0  0008A8B0 00000000 00000000 00089E1E
cmp.b   #$1A,D0
-■
```

Yes No ask

書　式　Y/N

機　能　Y/N ポーズ

Y か N の入力を促して、コマンドを一時中断します。

解　説　デバッガのコマンドを一時中断します。

継続する場合には Y を、中止する場合には N をキーインしてください。

コマンドラインのループ（¥）と併用します。

なお、⏎だけ押すと Y を入力することと同じになります。

例　Y/N コマンドの使用例を以下に示します。

```
-:D:L:Y/N:¥ ⏎      (複数のコマンドを連結して実行する場合には各コマンドを：で区切ります)
```

ダンプ実行

↓

逆アセンブル実行

↓

```
Y/N ? ■            処理継続の問い合わせ
Y/N ? Y            Y を入力すると処理を継続する
```

↓

ダンプ実行

↓

逆アセンブル実行

↓

```
Y/N ?              再度処理継続の問い合わせ
Y/N ? N            N を入力すると処理を打ち切る
-■
```

Display System Variables

書式	Z
機能	システム変数の表示

解説　Zコマンドは、全システム変数の現在値を表示します。
システム変数とは、頻繁に使用する値をZ0～Z9の変数で代用させて、いちいち、複雑な値を入力しなくても、デバッグをスムースに行えるようにしたものです。
使用方法については次ページのシステム変数の設定を参照してください。

例　Zコマンドの使用例を以下に示します。

```
-Z↵
Z0:00000000
Z1:00000000
Z2:00000000
Z3:00060500
Z4:00000000
Z5:00000000
Z6:00000000
Z7:00000000
Z8:00000000
Z9:00000000
-■

-D .Z3↵
00060500  FF00 0000 0000 0000 0000 0000 FFFF FFFF
00060510  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
00060520  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
00060530  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
00060540  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
00060550  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
00060560  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
00060570  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
-■
```

D. Z3と指定すると、システム変数3の現在値$60500からダンプが開始されます。

Set System Variable

書　式　Z<番号>=式

機　能　システム変数の設定

解　説　システム変数の値を設定します。
<番号>はシステム変数の番号で、0から9まで指定できます。
また、システム変数は、.（ピリオド）をつけてコマンドラインで参照することができます。

例　Zコマンドの使用例を以下に示します。

```
-Z ⏎ ──────────────── システム変数の表示
Z0:00000000
Z1:00000000
Z2:00000000
Z3:00000000
Z4:00000000
Z5:00000000
Z6:00000000
Z7:00000000
Z8:00000000
Z9:00000000
-■

-Z3=00060500 ⏎ ──────────── システム変数3に値を設定
-■

-Z ⏎ ──────────────── システム変数の表示
Z0:00000000
Z1:00000000
Z2:00000000
Z3:00060500
Z4:00000000
Z5:00000000
Z6:00000000
Z7:00000000
Z8:00000000
Z9:00000000
-■
```

Print expression (hex)

書　式　?＜式＞

機　能　計算式の結果の 16 進表示

解　説　＜式＞で指定した式の結果を 16 進数で表示します。
式中にスペースやタブを入れてはいけません。

例　?コマンドの使用例を以下に示します。

```
-?1F+A0↵
000000BF
-■
```

Print expression (dec)

書　　式　??<式>

機　　能　計算式の結果の10進表示

解　　説　<式>で指定した式の結果を10進数で表示します。
式中にスペースやタブを入れてはいけません。

例　??コマンドの使用例を以下に示します。

```
-??1A*2C ↵
1144
-■
```

Loop command line

書　式	¥
機　能	コマンドをくり返し実行します。
解　説	入力したコマンドラインの繰り返し実行を行います。 繰り返し実行を中断するときは、CTRL＋Cを押してください。
例	¥コマンドの使用例を以下に示します。

```
-:D:L:¥
```

（複数コマンドを連続して実行する場合には各コマンドを：で区切ります）

ダンプと逆アセンブルを、CTRL＋Cにより中断されるまで繰り返します。

Output Redirect

書　式　＞［<ファイル名>］
＞＞［<ファイル名>］

機　能　出力リダイレクト

解　説　＞，＞＞コマンドは、デバッガがコンソールに出力するメッセージをファイルにリダイレクトします。

＞コマンドは、指定されたファイルを新規作成してリダイレクト出力します。

また、＞＞コマンドは、指定されたファイルがすでに存在している場合、追記書き込みを行います。

<ファイル名>には、メッセージを保存したいファイルを指定します。

<ファイル名>が省略されると、現在行っているリダイレクト出力を中止します。

例　＞，＞＞コマンドの使用例を以下に示します。

```
-> debug.prn ⏎

-DL 000000 L20 ⏎
00000000  00FF0538 01FF0538 0003C136 0003C144         ...8...8..ﾁ6..ﾁD
00000010  0003C156 0003C170 0003C180 0003C194         ..ﾁV..ﾁp..ﾁ.ﾁ
-> ⏎
->> debug.prn ⏎

-D 000080 00009F
00000080  20FF 0538 21FF 0538 22FF 0538 23FF 0538     ..8!..8"..8#..8
00000090  24FF 0538 25FF 0538 26FF 0538 27FF 0538     $..8%..8&..8'..8
-> ⏎
```

debug. prn には、次の内容が格納されます。

```
-
00000000  00FF0538 01FF0538 0003C136 0003C144         ...8...8..ﾁ6..ﾁD
00000010  0003C156 0003C170 0003C180 0003C194         ..ﾁV..ﾁp..ﾁ.ﾁ
-
-
00000080  20FF 0538 21FF 0538 22FF 0538 23FF 0538     ..8!..8"..8#..8
00000090  24FF 0538 25FF 0538 26FF 0538 27FF 0538     $..8%..8&..8'..8
-
```

Command Logging

書　式	＞@［＜ファイル名＞］
機　能	入力コマンドのファイルへの保存

解　説

＞@コマンドは、キーボードよりデバッガに対して入力したコマンドを、ファイルにリダイレクト出力して保存します。

＜ファイル名＞には、リダイレクト出力するファイル名を指定します。

＜ファイル名＞を省略すると、現在行っているコマンドのファイル保存を中止します。

＞@コマンドでリダイレクト出力されたファイルは＜コマンドで使用することができます。

例

＞@コマンドの使用例を以下に示します。

```
->@ dbcom.prn⏎

-DL 000000 L20⏎
00000000  00FF0538 01FF0538 0003C136 0003C144            ...8...8..ﾁ6..ﾁD
00000010  0003C156 0003C170 0003C180 0003C194            ..ﾁV..ﾁp..ﾁ.ﾁ
-L 6800⏎
00006800        bra.w    $00007170
00006804        moveq    #$8F,D0
00006806        trap     #$0F
00006808        cmp.l    #$10870305,D0
0000680E        bcs.w    $00006FC4
00006812        moveq    #$AF,D0
00006814        trap     #$0F
00006816        move.l   #$00007304,D0
-@⏎

-■
```

この例では、dbcom. prn には以下のように入力コマンドが保存されます。

```
-DL 000000 L20
L 6800
>@
```

Input Redirect

書　　式　<<ファイル名>

機　　能　入力のリダイレクト

解　　説　<コマンドは、コマンドの投入をファイルから行います。
ファイルがEOFに達するまで継続します。
<ファイル名>には、デバッガのコマンドを記述したファイルの名前を指定します。
<コマンドで指定するファイルは、通常>@コマンドで出力されたファイルを使用します。

例　<コマンドの使用例を以下に示します。
ファイル dbcom. prn の内容が以下の通りだとします。

```
DL 000000 L20
L6800
```

そして、デバッガを起動して、<コマンドを実行すると以下のようになります。

```
-< dbcom.prn↵

-DL 000000 L20↵
00000000  00FF0538 01FF0538 0003C136 0003C144          ...8...8..ﾁ6..ﾁD
00000010  0003C156 0003C170 0003C180 0003C194          ..ﾁV..ﾁp..ﾁ.ﾁ
-L 6800↵
  00006800      bra.w    $00007170
  00006804      moveq    #$8F,D0
  00006806      trap     #$0F
  00006808      cmp.l    #$10870305,D0
  0000680E      bcs.w    $00006FC4
  00006812      moveq    #$AF,D0
  00006814      trap     #$0F
  00006816      move.l   #$00007304,D0
-■
```

OS Command

書式　![<コマンド名>]

機能　OSのコマンドを実行します。

解説　OS（Human68k）のコマンドを実行します。
<コマンド名>には、実行するOSのコマンドを指定します。
省略すると、COMMAND. Xを実行します。
COMMAND. Xからデバッガに戻るには、exitコマンドを実行してください。
環境変数pathを参照して、指定されたコマンドを検索するので、カレントディレクトリにないコマンドでも、コマンド名を指定するだけで実行できます。
もし、検索に失敗した場合はコマンド・プロセッサ（COMMAND. X）を呼び出して実行させます。
!コマンドによってOSのコマンドを実行するためには、そのための空きメモリが必要です。
この空きメモリのサイズはデフォルトで128Kバイト確保されます。
OSコマンドの実行のためのメモリが128Kバイトでは足りないか、または128Kより小さくしたい場合は、デバッガの起動時に／cスイッチにより空きメモリのサイズを指定することができます。
また、実行したコマンドが、デバッグ対象プログラムやその環境に対して何か影響を与える可能性があることに注意してください。

例　!コマンドの使用例を以下に示します。

```
X68k Debugger v2.00 Copyright 1987,88,89,90 SHARP/Hudson
-! DIR A: ↵


 XCシステム#1              A:¥
    16 ファイル         1066K Byte 使用中        155K Byte 使用可能
 ファイル使用量         177K Byte 使用
CONFIG             SYS        469  90-05-05  12:00:00
KEY                SYS        712  87-05-15  12:00:00
USKCG              SYS       8300  90-05-05  12:00:00
STARTUP            ENV         48  90-05-05  12:00:00
AUTOEXEC           BAT        678  90-05-05  12:00:00
COMMAND            X        28026  90-05-05  12:00:00
VS                 X        97540  87-05-15  12:00:00
ICONDATA           VS       39168  89-02-10  12:00:00
CLIP               VS           0  87-05-15  12:00:00
CONFIG                 <dir>       90-05-05  12:00:00
SHELL                  <dir>       90-05-05  12:00:00
SYS                    <dir>       90-05-05  12:00:00
BIN                    <dir>       90-05-05  12:00:00
ASK                    <dir>       90-05-05  12:00:00
HIS                    <dir>       90-05-05  12:00:00
ETC                    <dir>       90-05-05  12:00:00

デバッガに戻ります！何かキーを押してください
-■
```

6.5 DBエラーメッセージ一覧

エラーメッセージ	意　味
Bad breakpoint number (0-9)	ブレークポイントの番号が範囲外です
Breakpoint list or '*' expected	ブレークポイントリストまたは*を指定していません
no break pointer	ブレークポイントがありません
break pointer over	すでにブレークポイントがいっぱいです
Undefined symbol<シンボル>	未定義シンボルです
Expression error	式表現に誤りがあります
undefined instruction<コード>	未定義命令を実行しようとしました
zero divide	0で除算しました
chk instruction	chk 命令で実行しました
trapv instruction	trapv 命令で実行しました
privilege violation	特権違反をしました
Exceptional Abort By bus error By Memory Access of<アドレス>	バスエラーです
Exceptional Abort By address error By Memory Access of <アドレス>	アドレスエラーです
double dxception in system status display	例外処理中にエラーが発生しました
offset overs	オフセットが範囲外です
fatal error	致命的エラーです
File create error	ファイルがつくれません
Device write error	デバイス書き込みエラーです
No symbol file	ファイルにシンボルがありません
File read error	ファイルが読み込めません
Device error	デバイスエラーです
Command error<コマンド>	コマンドエラーです

6.5 DBエラーメッセージ一覧

エラーメッセージ	意　味
Bad parameter	パラメーターに誤りがあります
data too long	データが長すぎます
size over	サイズが大きすぎます
no memory check pointer	メモリチェックポイントがいっぱいです
memory check pointer over	すでにメモリチェックポイントの番号が範囲外です
Bad checkpoint number (0-9)	メモリチェックポイントの番号が範囲外です
Checkpoint list or '*' expected	メモリチェックポイントリストまたは*を指定していません

第7章

アーカイバ

アーカイバとは
アーカイバが使用するファイル
使用書式と起動方法
アーカイバのスイッチ
アーカイバの使用例
AR エラーメッセージ一覧

アーカイバという名前は、書庫を意味します。
つまり、多くのファイルを書庫で扱うように管理するわけです。
書庫の中、すなわち、1つのアーカイブファイルの中に、関連したファイルをいくつも登録することができます。

アーカイバ XArchiver の機能は、次の通りです。

- ●ファイルの登録
- ●ファイルの更新
- ●ファイルの削除
- ●ファイルの抽出
- ●ファイルの表示

7.1 アーカイバとは

プログラムの開発の過程で、複数個の文書ファイルが作成される場合があります。
このような複数個のファイルを、ファイルの種類ごとに、ひとまとめにして管理したいような場合にアーカイバを使用することができます。

たとえば、

```
test1. txt
test2. txt
test3. txt
  :
test9. txt
```

という９個のテキストファイルがあったとき、これらをひとまとめにして１個のファイル（これをアーカイブファイルといいます）で管理することができます。

```
test1. txt ┐
test2. txt │
test3. txt ├ ➡ x68k. a(「a」はアーカイブファイルの拡張子です)
  :        │
test9. txt ┘
```

このように、ファイルをひとまとめにしておくことにより、DIR の表示は見やすくなり、ファイルが整理されるので、管理しやすくなります。
このような整理や管理を一手に引き受けるのがアーカイバです。

7.2 アーカイバが使用するファイル

アーカイバは次の種類のファイルを使用します。

●アーカイブファイル

複数のファイルの書庫の名称だとご理解ください。
つまり、1つのアーカイブファイルという入れ物の中に、複数のファイルを含むわけです。
アーカイブファイルは、ディレクトリからみると、1つのファイルとしてみなされます。
そこで、オペレーティングシステムからは、1つのファイルとして扱われます。
コピーや消去なども通常のファイルと同じように行えます。
アーカイブファイルの拡張子は、自動的にaとなります。

●テキストファイル

ソースプログラムなど、テキストファイルをアーカイブファイルにまとめておくことができます。
テキストファイルを扱うときは、必ず/aスイッチによって、アスキーモードに設定してください。
整理されるファイル名の拡張子を省略したとき、/aスイッチをつけると拡張子はsとみなされます。

7.3 使用書式と起動方法

起動する前に確認すること

●アーカイバが扱うファイルのパス名は正しいですか?

新しくアーカイブファイルを作成するときは、コマンドラインで名称を指定するだけでけっこうです。
しかし、すでにアーカイブファイルがあるときは、／l スイッチでその中の状態を見ておきましょう。

```
AR　／a　／v　／l ascii ↵
```

とすると、ascii. a の中の状態が表示されます。

```
X68k Archiver v1.00 Copyright 1987 SHARP／Hudson
  2333 1987-04-08 13：48：30 artest. s
  2333 1987-04-08 13：48：30 aaa. s
```

●パス上にアーカイバの起動に必要な空きがありますか?

特に、新しくアーカイブファイルを作成するときは、これからアーカイブしようとしているファイルのコピーを作るのですから、ちょうどコピーができる程度の空き領域が必要です。

●同一コマンドライン上でテキストファイルとオブジェクトファイルを同時に指定していませんか?

1つのアーカイブファイルの中で、テキストファイルとオブジェクトファイルなどのバイナリファイルを混在して扱うことはできません。
また、テキストファイルの拡張子はｓとし、オブジェクトファイルの拡張子はｏとしてください。
なお、オブジェクトファイルをまとめて1つのファイル（ライブラリファイル）にするためのものにライブラリアン（LIB. X）が用意されていますので、オブジェクトファイルを扱う場合はライブラリアンを使用してください。
ライブラリアンについては第8章で説明していますので、そちらをご覧ください。

アーカイバの起動方法と書式

コマンドラインで次のような書式で入力します。

```
AR［&lt;スイッチ&gt;］&lt;アーカイブファイル名&gt;&lt;ファイル名&gt;
 ［&lt;ファイル名&gt;…］
```

ファイル名は複数個指定できます。
アーカイブファイルの名称は、スイッチに引き続いて指定します。
アーカイブファイル名の拡張子を省略するとａとみなされます。

また、テキストファイルを扱うときは、必ず／ａスイッチを指定します。
新規にアーカイブファイルを作成するときは、ここでそのアーカイブファイルの名称を指定するだけでけっこうです。

7.3 使用書式と起動方法

正しくコマンドラインが入力されると、ただちにアーカイバが起動します。

もし誤ったスイッチを指定すると次のようなヘルプメッセージが表示されます。
コマンドラインを確認し直してください。

```
X68k Archiver v1.00 Copyright 1987 SHARP/Hudson
使用法：ar ［スイッチ］ ライブラリ ［ファイル・・・］
         /a      ＡＳＣＩＩファイルモード
         /b      バックアップファイルの作成
         /u      ファイルの更新（デフォルト）
         /x      ファイルの取り出し
         /d      ファイルの削除
         /l      リスト出力
         /i file インダイレクトファイルの指定
         /v      バーボーズモード
```

誤って存在しないアーカイブファイルを指定して(新規作成以外)操作を行うと、

```
archive file not found
```

と表示されます。

アーカイブされるファイルが存在しないときは、

```
file not found<ファイル名>
```

と表示されます。
ディレクトリ、または対象となるファイルを確認してください。

7.4 アーカイバのスイッチ

アーカイバのスイッチには、次のものがあります。

スイッチ	機　能
／a	テキストファイルの指定
／b	バックアップファイルの作成
／d	ファイルの削除
／i	インダイレクトファイルの指定
／l	アーカイブファイルの内容表示
／u	ファイルの登録・更新指定
／v	バーボーズモードの指定
／x	ファイルの抽出

／a　テキストファイルの指定

書式

／a

機能

テキストファイルの使用を宣言します。

解説

ソースプログラムなどのテキストファイルの使用を宣言します。
1つのアーカイブファイルでは、テキストファイルとオブジェクトファイルを混在して扱うことはできませんので、後続の<整理されるファイル名>に、オブジェクトファイルを指定することはできません。
テキストファイルをアーカイブするときは、他のスイッチとともに、必ずこのスイッチを指定してください。

例

／a スイッチの使用例を以下に示します。

```
AR　／a　／u archive source1 source2 source3 ⏎
```

アーカイブファイル archive に登録されるファイル source1～source3 はすべてテキストファイルです。

/b バックアップファイルの作成

書式 ／b

機能 バックアップファイルを作成します。

解説 アーカイブファイルを更新したときに、更新前の内容のバックアップファイルを作成します。
なお、バックアップファイルの拡張子はbakとなります。

例 ／bスイッチの使用例を以下に示します。

```
AR /a /b archive file1 file2
```

アーカイブファイルarchiveを更新する前に、バックアップファイルも作成します。
バックアップファイル名は次のようになります。

archive. a ⟶ archive. bak

/d　ファイルの削除

書式　/d

機能　指定されたファイルを、アーカイブファイルから削除します。

解説　指定されたファイルを、アーカイブファイルから削除します。

例　/dスイッチの使用例を以下に示します。

```
AR　/d archive file1. s⏎
```

アーカイブファイル archive の中に file1. s が存在した場合、これを削除します。

/i インダイレクトファイルの指定

書式 /i

機能 インダイレクトファイルを指定します。

解説 アーカイバ操作時に使用するインダイレクトファイルを指定します。
インダイレクトファイルには、スイッチ、アーカイブファイル名、テキストファイル名またはオブジェクトファイル名などを登録します。
この指定は、入力するスイッチ、ファイル名などが長い場合や、入力するパラメータが同じで、これを何度も繰り返して使用する場合などに効果的です。
インダイレクトファイルが指定されると、アーカイブはその内容を参照して各処理を行います。
一度インダイレクトファイルを作成しておけば、アーカイバ操作時ごとにスイッチやファイル名を指定する必要がないので、アーカイブの作業効率を向上させることができます。
なお、インダイレクトファイルは、エディタを使用して作成します。

例 /i スイッチの使用例を以下に示します。

```
AR  /i indirect ↵
```

インダイレクトファイル indirect の内容

```
/a  /u
archive
source1 source2 source3 source4
```

アーカイブを起動すると、テキストファイル source1～source4 が、アーカイブファイルに登録、更新されます。

/l アーカイブファイルの内容表示

書　式　/l

機　能　アーカイブファイルの内容を表示します。

解　説　アーカイブファイルに登録されている全ファイル名を表示します。
また、/lスイッチを指定する場合、/vスイッチ以外の他のスイッチや整理するファイルを指定してはいけません。

例　/lスイッチの使用例と全ファイル名の表示形式を以下に示します。

```
AR  /l archive ⏎
```

アーカイブファイル archive に登録済みの全ファイル名を表示します。

```
X68K Archiver v1.00 Copyright 1987 SHARP/Hudson
file1. s
file2. s
file3. s
  :
```

/v スイッチと併用するとファイル内容がくわしく表示されます。

／u ファイルの登録・更新指定

書　式　／u

機　能　ファイルをアーカイブファイルに新規に登録するか、または更新することを宣言します。

解　説　このスイッチが指定されると、各ファイルがアーカイブファイルに登録されるか、またはアーカイブファイルに登録済みの同一ファイルを更新することを示します。
このスイッチは指定しなくても差しつかえありません。

例　／u スイッチの使用例を以下に示します。

```
AR ／a ／u archive source1 source2 source3 ↵
```

テキストファイル source1〜source3 は、アーカイブファイル archive に登録されます。

/V　バーボーズモードの指定

書　式　／v

機　能　バーボーズモードを指定します。

解　説　バーボーズモードを指定すると、アーカイバ操作時の情報が表示されます。

例　／v スイッチの使用例と、表示される情報の内容を以下に示します。

```
AR ／a ／v ／u archive text1 ↵
```

テキストファイル text1 をアーカイブファイル archive に登録するとともに、登録処理中の情報を表示します。

```
X68K Archiver v1.00 Copyright 1987 SHARP／Hudson
include text1
```

／x　ファイルの抽出

書　式	／x
機　能	指定されたファイルを、アーカイブファイルから抽出します。
解　説	指定されたファイルを、アーカイブファイルから抽出します。
例	／x スイッチの使用例を以下に示します。

```
AR  /a  /x archive file1.s ↵
```

アーカイブファイル archive から file1. s を抽出し、テキストファイル file1. s を作成します。

7.5 アーカイバの使用例

```
AR　/a test temp. s↵
```

test. aという名称のアーカイブファイルにtemp. sというテキストファイルを登録します。
このとき、次のように3つの場合があります。

1. test. aが新規ファイル
   新しくtest. aというアーカイブファイルを作成してそこにtemp. sを登録します。

2. test. aが既存のアーカイブファイルでtemp. sは新規ファイルです。
   temp. sを新規に登録します。

3. test. aが既存のアーカイブファイルでかつtemp. sはすでに一度アーカイブされています。
   temp. sを更新します。

上記2の操作例を次に示します。

```
A>AR　/a test. a temp. s↵
```

temp. sがほんとうに登録されたかどうか、確認してみましょう。

```
A>AR　/a　/l test. a↵
```

7.5 アーカイバの使用例

```
X68k Archiver v1.00 Copyright 1987 SHARP／Hudson
temp1. s
temp2. s
temp3. s
temp4. s
temp. s
```

7.6 ARエラーメッセージ一覧

エラーメッセージ	意　味
Abort: Indirect mode error	インダイレクトファイル指定のエラーです。
Abort: Indirect file not found	インダレクトファイルがありません。
can not create file	テンポラリーファイルが作れません。
archive file not found	アーカイブファイルがありません。
no file name	ファイル名が指定されていません。
not archive file	アーカイブファイルではありません。
can not delete＜ファイル名＞	ファイルを削除できません。
file not found＜ファイル名＞	ファイルがありません。
not object file＜ファイル名＞	オブジェクトファイルではありません。
Abort Device full	ディスクがいっぱいです。

第8章

ライブラリアン

ライブラリアンが使用するファイル
使用書式と起動方法
ライブラリアンのスイッチ
ライブラリアンの使用例
LIB エラーメッセージ一覧

プログラムの開発は、機能ごとのモジュールに分割して行うことが多いので、1つの実行可能プログラムを作成する過程で数多くのオブジェクトファイルが作成されます。
このような複数個のオブジェクトファイルをライブラリアンによって、機能ごとにオブジェクトをまとめたファイルをライブラリファイル（拡張子はl）といいます。
ライブラリアンはアーカイバと機能がよく似ていますが、以下の点が違います。

- ●ライブラリファイルはアーカイブファイルより高速にリンクできます。
- ●オブジェクトファイルのみ登録できます。
  アーカイバのようにテキストファイルは登録できません。
- ●ライブラリファイルの先頭に、登録されているオブジェクトファイルごとのシンボル情報や各モジュールの管理情報を格納したヘッダが加えられます。

つまり、ライブラリアンは、リンクする複数のオブジェクトを管理するのが主な用途といえます。
それに対してアーカイバは、ソースプログラムを管理するのが主な用途となります。
ライブラリアンの機能は以下の通りです。

- ●ファイルの登録
- ●ファイルの更新
- ●ファイルの削除
- ●ファイルの抽出
- ●ファイルの表示

8.1 ライブラリアンが使用するファイル

ライブラリアンは次の種類のファイルを使用します。

●ライブラリファイル

複数のオブジェクトファイルをまとめたファイルです。
オブジェクトファイルしか格納できません。
また、ライブラリファイルは直接リンクできます。
ライブラリアンは、オブジェクトファイルをまとめるときに、そのシンボル情報と各モジュールの管理情報をライブラリファイルの先頭に格納します。
そのため、アーカイブファイルと比べて高速にリンクできます。
ライブラリファイル自体は、通常ファイルと同じ扱いができます。
つまり、オペレーティングシステムからは1つのファイルとして扱われます。コピーや消去なども通常のファイルと同じように行えます。
ライブラリファイルの拡張子は、自動的に1になります。

●オブジェクトファイル

アセンブラがソースプログラムをアセンブルして作成されるファイルです。
このファイルをライブラリアンがまとめてライブラリファイルを作成します。

8.2 使用書式と起動方法

起動する前に確認すること

●ライブラリアンが扱うファイルのパス名は正しいですか？

新しくライブラリファイルを作成する時は、コマンドラインで名称を指定するだけでけっこうです。
しかし、すでにライブラリファイルがあるときは、/lスイッチでその中の状態を見ておきましょう。

●パス上にライブラリアンの起動に必要な空きがありますか？

特に、新しくライブラリファイルを作成する時は、これから登録しようとしているファイルのコピーを作るのですから、ちょうどコピーができる程度の空き領域が必要です。

ライブラリアンの起動方法と書式

コマンドラインで次のような書式で入力します。

```
LIB [〈スイッチ名〉] 〈ライブラリファイル名〉
  〈ファイル名〉 [〈ファイル名〉 …]
```

ファイル名は複数個指定できます。
新規にライブラリファイルを作成するときは、ここでそのライブラリファイルの名称を指定するだけです。
正しくコマンドラインが入力されると、ただちにライブラリアンが起動します。
もし誤ったスイッチを指定すると次のようなヘルプメッセージが表示されます。コマンドラインを確認して直してください。

```
X68k Librarian v1.00 Copyright 1990 SHARP/Hudson
使用法：lib [スイッチ] ファイル [ファイル・・・]
        /m nn    最大シンボル数(201<nn<65536)
        /b       バックアップファイルの作成
        /u       ファイルの更新（デフォルト）
        /x       ファイルの取り出し
        /d       ファイルの削除
        /l       リスト出力
        /i file  インダイレクトファイルの指定
        /v       バーボーズモード
```

また、誤って存在しないライブラリファイルを指定して(新規作成以外)操作を行った時も同様です。

登録されるファイルが存在しないときは、以下のようなメッセージを表示します。

```
source file open error＜ファイル名＞
```

ディレクトリ、または対象となるファイルを確認してください。

8.3 ライブラリアンのスイッチ

ライブラリアンのスイッチには、次のものがあります

スイッチ	機　能
／b	バックアップファイルの作成
／d	ファイルの削除
／i	インダイレクトファイルの指定
／l	ライブラリファイルの内容表示
／m	シンボルの最大個数の指定
／u	ファイルの登録・更新指定
／v	バーボーズモードの指定
／x	ファイルの抽出

/b　バックアップファイルの作成

書　式　／b

機　能　バックアップファイルを作成します。

解　説　ライブラリファイルの中のファイルに対して修正を行ったとき、ライブラリファイルのバックアップファイルを作成し、ここに更新前の内容を保存します。
なお、バックアップファイルの拡張子は、bak となります。

例　／b スイッチの使用例を以下に示します。

```
LIB  /b  test srcl ⏎
```

ライブラリファイル test.l の中の srcl.o を更新し更新後のライブラリファイルを test.l、更新前のバックアップファイルを test. bak とする。

／d　ファイルの削除

書　式	／d
機　能	指定されたファイルを、ライブラリファイルより削除します。
解　説	指定されたファイルを、ライブラリファイルより削除します。
例	／d スイッチの使用例を以下に示します。

```
LIB  /d userdef funcl ↵
```

ライブラリファイル userdef. l の中にオブジェクトファイル funcl. o が存在した場合、これを削除します。

/i インダイレクトファイルの指定

書式　/i <ファイル名>

機能　インダイレクトファイルを指定します。

解説　ライブラリアン操作に使用するインダイレクトファイルを指定します。
インダイレクトファイルには、スイッチ、ライブラリファイル名、オブジェクトファイル名などを登録します。
この指定は、入力するスイッチ、ファイル名などが長い場合や、入力するパラメータが同じで、これを何度も繰り返して使用する場合などに効果的です。
インダイレクトファイルを指定されると、ライブラリアンはその内容を参照して各処理を行います。
一度インダイレクトファイルを作成しておけば、ライブラリアン操作時ごとにスイッチやファイル名を指定する必要がないので、作業効率を向上させることができます。
なお、インダイレクトファイルは、エディタを使用して作成してください。

例　/i スイッチの使用例を以下に示します。

```
LIB /i indirect ↵
```

インダイレクトファイル indirect の内容

```
/u
test
src1  src2  stc3  src4  src5  src6
```

ライブラリアンを起動すると、オブジェクトファイル src1. o～src6. o がライブラリファイル test. l に登録、更新されます。

／l　ライブラリファイルの内容表示

書　式　／l

機　能　ライブラリファイルの内容を表示します。

解　説　ライブラリファイルに登録されている全ファイル名を表示します。
また、／l スイッチを指定する場合、／v スイッチ以外のスイッチや整理するファイルを指定してはいけません。

例　／l スイッチの使用例と全ファイル名の表示形式を以下に示します。

```
LIB ／l tst
```

ライブラリファイル tst. l に登録済みの全ファイル名を表示します。

```
X68k Librarian v1.00 Copyright 1990 SHARP/Hudson
file1.o
file2.o
file3.o
   :
   :
   :
```

／m　シンボルの最大個数の指定

書式　／m<nn>

機能　シンボルの最大個数を指定します。

解説　定義可能なシンボルの最大個数を指定します。
定義可能なシンボルの個数は、以下の範囲です。

201＜nn＜65536

なお、／m スイッチを指定しない場合は、2000 個になります。

例　／m スイッチの使用例を以下に示します。

```
LIB  /m4000  tst.l  src.o
```

定義可能なシンボルは、4000 個です。

シンボル 1 個当たりの記憶域のサイズは、約 16 バイト必要です。

/u ファイルの登録、更新指定

書式　/u

機能　ファイルをライブラリファイルに新規に登録するか、または更新することを宣言します。

解説　このスイッチが指定されると、各オブジェクトファイルがライブラリファイルに登録されるか、またはライブラリファイルに登録済みの同一ファイルを更新することを示します。
このスイッチは指定しなくても差しつかえありません。

例　/uスイッチの使用例を以下に示します。

```
LIB /u test src1 src2 src3⏎
```

オブジェクトファイルsrc1.o～src3.oを、ライブラリファイルtest.lに登録します。

/v　バーボーズモードの指定

書式　/v

機能　バーボーズーモードを指定します。

解説　バーボーズモードを指定すると、ライブラリアン操作時の情報が表示されます。

例　/v スイッチの使用例と、表示される情報の内容を以下に示します。

```
LIB /v /u test object1
```

オブジェクトファイル object1.o をライブラリファイル test.l に登録するとともに、登録処理中の情報を表示します。

```
X68k Librarian v1.00 Copyright 1990 SHARP/Hudson
include object1
```

/X ファイルの抽出

書　式　／x

機　能　指定されたファイルを、ライブラリファイルから抽出します。

解　説　指定されたファイルを、ライブラリファイルから抽出します。

例　／x スイッチの使用例を以下に示します。

```
LIB /x test src1 ↵
```

ライブラリファイル test. l から src1 を抽出し、オブジェクトファイル src1. o を作成します。

8.4 ライブラリアンの使用例

ライブラリファイル test. l にオブジェクトファイル temp. o を登録するには、

```
LIB  /u  test  temp↵
```

を実行します。
このコマンドラインの意味は、test. l の状態により以下の 3 つの場合に分けられます。

1. test. lが新規ファイルの場合
   新しくライブラリファイルtest. lを作成し、それにオブジェクトファイルtemp. oを登録します。
2. test. lが既存のファイルで、temp. oが未登録の場合test. lにtemp. oを登録します。
3. test. lが既存ファイルでtemp. oも登録してある場合test. lに登録されているtemp. oを更新します。

ライブラリファイルに登録されているオブジェクトを知りたいときは、以下のコマンドラインを実行します。

```
LIB  /l test  ↵
```

8.5 LIB エラーメッセージ一覧

エラーメッセージ一覧	意　味
Abort : Indirect mode error	インダイレクトの指定の誤り
Abort : Illegal option error	スイッチの誤り
Abort : Not library file	ライブラリファイルの指定の誤り
Abort : Indirect file not found	インダイレクトファイルが見つかりません
Abort : System error	システムエラーが発生しました
Abort : Illegal file error	ファイルの指定の誤り
Abort : file read error	ファイルの読み込みに失敗した
Abort : file write error	ファイルの書き込みに失敗した
source file open error＜ファイルネーム＞	登録・更新されるファイルがオープンできません
destination file open error＜ファイルネーム＞	抽出されるファイルがオープンできません
LIB : Internal File Open Error	ライブラリアンが内部で使用するファイルがオープンできません
Not enough memory	メモリ不足です
illeagl option＜指定スイッチ＞	スイッチの指定が違います

第 9 章

コンバータ

コンバータが扱うプログラム形式
使用書式と起動方法
コンバータのスイッチ
コンバータの使用例
CV エラーメッセージ一覧

コンバータ XConverter は、リンカが生成した実行可能プログラム (x 形式、拡張子が x であるもの) を他の実行可能プログラムの形式に変換します。

変換される形式には、次の 2 つがあります。

r 形式

フルリロケータブルの実行可能プログラム

z 形式

絶対アドレス指定の実行可能プログラム

リンカが生成した x 形式の実行可能プログラムは、Human68k が実行するための管理情報を持っています。
具体的には、実行可能プログラムは、Human68k によってロードされるときにメモリ内へ適切に再配置されます。
つまり、そのままメモリにロードされるのではありません。
このため、再配置のための処理の時間がわずかですが必要ですし、再配置のための情報がファイル上に書かれているため、サイズも大きくなっています。

9.1 コンバータが扱うプログラム形式

Human68k には、次の3つの実行プログラム形式があります。

x 形式

x 形式は、リンカが出力する実行可能プログラム形式です。
実行に先立って、ローディング時にメモリに再配置されます。
x 形式のプログラムファイルには、このため再配置の情報が含まれています。

r 形式

r 形式は、フルリロケータブル実行可能プログラム形式です。
フルリロケータブルというのは、メモリ内のどの位置にロードしても再配置なしに実行が可能であるということです。
r 形式の実行可能プログラムでは、このように、メモリのどの位置にロードしても実行が可能です。
（ただし、システムや他のアプリケーションが占有しているメモリ領域には、ロードできません。）
分岐命令やサブルーチンコール命令で使用する変位(ディスプレイスメント)はプログラム内相対が可能となっているので、MPU68000 では、プログラム全体をどの位置にロードしても実行できるようなプログラムを作成することができます。
このような形式で書かれたファイルが r 形式のファイルです。

z 形式

固定番地にロードされないと実行できない形式のファイルです。

9.2 使用書式と起動方法

起動する前に確認すること

- 変換する前のx形式のファイルが存在してますか?

コンバータの起動方法と書式

コマンドラインから次のような書式で入力します。

CV［<スイッチ>］<変換前のx形式ファイル>［<変換後のrまたはz形式のファイル名>］

正しくコンバータが起動されると、ただちにコンバートを開始します。
もし、誤って使用すると次のようなヘルプメッセージが表示されます。

```
X68k File Converter v2.00 Copyright 1987,88,89,90 SHARP/Hudson
使用法：cv ［スイッチ］ ソースファイル ［オブジェクトファイル］
        /r             Ｒファイルの作成（デフォルト）
        /rn            Ｒファイルの作成（エラー無視）
        /z[address]    Ｚファイルの作成
        /h[address]    ＨＥＸファイルの作成
        /m[address]    Ｓフォーマットファイルの作成
        /b[address]    ＢＳＳアドレス指定（Ｒファイル以外）
        /o             奇数ファイルと偶数ファイルに分ける
```

誤って存在しないファイルを指定すると

```
読み込みエラー
```

が表示されます。
ディレクトリを確認してください。

コンバータに使用するファイル

コンバータに使用するファイルは、

- 入力に、x形式の実行可能プログラムファイル
- 出力に、r形式またはz形式のプログラムファイル

です。

これらの出力ファイルの指定は、コンバータ起動のときのスイッチで行います。

9.3 コンバータのスイッチ

コンバータ XConverter には次のスイッチがあります。

スイッチ	機　能
／b	BSS セクション先頭アドレスの指定
／h	インテル HEX フォーマットの作成
／m	モトローラ S フォーマットの作成
／o	奇数アドレスと偶数アドレスに分割
／r	フルリロケータブル実行形式への変換
／rn	フルリロケータブル実行形式への変換
／z	絶対アドレス指定実行形式への変換

/b　BSSセクション先頭アドレスの指定

書式　/b<address>

機能　BSSセクションの先頭アドレスを指定します。

解説　本スイッチを指定すると実行可能プログラム（拡張子x）を、絶対アドレス指定実行可能形式へ変換する際にBSSセクションの先頭アドレスを指定することができます。

例　/bスイッチの使用例を以下に示します。

```
CV　/z60000　/b68000 sample ⏎
```

x形式の実行可能プログラム（ファイル名sample. x）は絶対アドレス指定実行形式（ロードアドレス$60000番地）へと変換します。
その時BSSセクションが$68000番地から始まるように、設定されます。
/bスイッチを指定しないと、BSSセクションはDATAセクションに連続して配置されます。
/bスイッチを指定する場合、BSSセクションがDATAセクションと重なるように指定することはできません。
この機能は、ROM上で動作させるためにコード領域とワーク領域を分離しなければならないプログラムを作成する場合に指定します。

/h インテルHEXフォーマットの作成

書式　／h<address>

機能　インテル HEX 形式のファイルを作成します。

解説　／h スイッチを指定すると、実行可能ファイルをインテル HEX フォーマット形式にファイル変換して出力します。
<address>には、ロードされる絶対番地を指定します。

インテル HEX フォーマットは可読な文字で構成されていて、制御文字を含みません。
また、チェックの情報も含まれています。

例

```
CV  /h7000 sample1 ↵
```

address 情報の下 4 ケタはファイル中に、上 2 ケタは 1 つ目のファイルの拡張子になります。
たとえば、$86000～$A5000 までを実行すると、

sample 1. h08　→ 6000～f f f f
sample 1. h09　→ 0000～f f f f
sample 1. h0A　→ 0000～5000

上記 3 つのファイルが作られます。
なお拡張子の 2 ケタは、アドレスとは無関係に 1 づつ加算されていきます。

注：1 つのインテル HEX フォーマットの大きさには限界があります。
最大で 64K バイトまでです。
そこで、64K バイト以上を超えると複数のファイルに分割して出力されます。
sample1. h00
sample1. h01

／m　モトローラSフォーマットの作成

書式　／m＜address＞

機能　モトローラＳフォーマット形式のファイルを作成します。

解説　本スイッチを指定すると、実行可能ファイルをモトローラＳフォーマット形式にファイルを変換して出力します。
＜address＞には、ロードされる絶対番地を指定します。

モトローラＳフォーマットは可読な文字で構成されていて、制御文字を含みません。
また、チェックの情報も含まれています。

例

```
CV　／m6000 sample1 ⏎
```

x 形式の実行可能プログラム(ファイル名 sample1. x)は、6000 番地へロードする指示情報を含んだモトローラＳフォーマットのファイル(ファイル名 sample1. m)に変換されます。

/o 奇数アドレスと偶数アドレスに分割

書式　／o

機能　インテルHEXフォーマットやモトローラSフォーマットを出力する際に奇数アドレスと偶数アドレスとに分けて出力します。

解説　／hスイッチ（インテルHEXフォーマット出力）や／mスイッチ（モトローラSフォーマット出力）を指定するときに、この／oスイッチを一緒に指定すると、指定された実行可能ファイルを奇数アドレスと偶数アドレスに分けてファイルを出力します。

／oスイッチは必ず／hスイッチか／mスイッチと同時に指定してください。

また出力するファイル名は、指定されたソースファイル名の拡張子を以下のように置き換えたものとなります。

	インテルHEX	モトローラS
ソースファイル名	TEST.X	TEST.X
偶数ファイル名	TEST.H00	TEST.M0
奇数ファイル名	TEST.H01	TEST.M1

例　／oスイッチの使用例を以下に示します。

```
CV  /o  /m0  test
```

この例では、test.xを偶数アドレスと奇数アドレスで分割し、モトローラSフォーマットで出力します。

出力されるファイル名は、test.m0とtest.m1です。

/r　フルリロケータブル実行形式への変換

書　式　/r

機　能　フルリロケータブル実行形式への変換をします。

解　説　/r スイッチを指定すると、x 形式の実行可能プログラム(拡張子 x)をフルリロケータブル実行形式（拡張子 r）へと変換します。
ただし、当該プログラム内で固定番地を使っていた場合、これをフルリロケータブル実行形式に変換することはできません。

例　/r スイッチの使用例を以下に示します。

```
CV　/r sample1 sample2 ⏎
```

x 形式の実行可能プログラム(ファイル名 sample1. x)はフルリロケータブル実行形式（sample2. r）へと変換されます。

```
CV　/r sample3 ⏎
コンバートできません
```

sample3 では固定番地が使われていたために、コンバートは失敗しました。

/rn フルリロケータブル実行形式への変換

書式　／rn

機能　フルリロケータブル実行形式への変換を指定します。

解説　／rn スイッチを指定すると、x 形式の実行可能プログラム(拡張子 x)をフルリロケータブル実行形式(拡張子 r)へと変換します。
ただし、コンバート処理時に発生したエラーは無視します。

例　／rn スイッチの使用例を以下に示します。

```
CV　／rn sample1 sample2 ↵
```

x 形式の実行可能プログラム(ファイル名 sample1. x)はフルリロケータブル実行形式(ファイル名 sample2. r)へと変換されます。
このとき、エラーは無視されます。

/z　絶対アドレス指定実行形式への変換

書　式　／z<address>

機　能　絶対アドレス指定実行形式への変換を指定します。

解　説　本スイッチを指定すると、x形式の実行可能プログラム(拡張子x)を、絶対アドレス指定実行形式(拡張子z)へと変換します。
<address>には、プログラムのロードアドレスを指定します。

例　／zスイッチの使用例を以下に示します。

```
CV　／z60000 sample1 sample2 ⏎
```

x形式の実行可能プログラム(ファイル名sample1. x)は絶対アドレスの指定実行形式(ロードアドレス$60000番地、ファイル名sample2. z)へと変換されます。したがって、sample2. zは$60000番地以外にロードすることはできません。

9.4 コンバータの使用例

```
CV  /z60000 pp⏎
```

pp. x を$60000 から実行するファイル(pp. z)に変換します。

9.5 CVエラーメッセージ一覧

スイッチが不正です
メモリが不足です
読み込みエラー
書き込みファイルがオープンできません
書き込みエラー
奇数アドレスを指定しました。even 補正します
セクションがかさなりました
正しい X ファイルではありません
コンバートできません

第2部

リファレンス マニュアル

第1章

アセンブラ

アセンブラプログラムの構成要素
アドレス形式
リロケータブルなプログラム作成
アセンブラ擬似命令

この章は、MPU68000マクロアセンブラの文法についてのリファレンスです。

内容は次の通りです。

1.1　アセンブラプログラムの構成要素

アセンブラプログラムの作成で使用する文字セット、ステートメントの種類と書式、識別名などについて解説します。

1.2　アドレス形式

命令の実行で必要となる実効アドレスの指定形式を、例を用いて解説します。

1.3　リロケータブルなプログラムの作成

リロケータブル(再配置可能)なオブジェクトモジュールを作成するときに必要な注意事項について解説します。

1.4　アセンブラ擬似命令

アセンブラ擬似命令の機能と種類を、例を用いて解説します。

1.1 アセンブラプログラムの構成要素

この節では、アセンブラプログラムの構成要素を以下の項目ごとに説明します。

- 文字セット　(1. 1. 1)
- ステートメントの種類　(1. 1. 2)
- ステートメントの書式　(1. 1. 3)
- 識別名　(1. 1. 4)
- 定　数　(1. 1. 5)
- 演算子　(1. 1. 6)

1.1.1 文字セット

ソースプログラムのコーディングに使える文字は次の通りです。

- 英字　A～Z, a～z の 52 文字
- 数字　0～9 の 10 文字
- 特殊文字　（空 白）!　"　#　$　%　&　'　(　)
  *　+　,　－　.　／　:　;　<　=　>　?　@
  ¥
- 漢字　シフト JIS コード

注 ： 漢字は引用符の中か、コメントに使用してください。

1.1.2 ステートメントの種類

ソースプログラムは、ステートメントで構成します。

ステートメントは、3 種類に分けられます。

1. 実行命令ステートメント
2. アセンブラ擬似命令ステートメント

3.　コメントステートメント

1.　実行命令ステートメント

実行命令ステートメントは、MPU68000の機械語命令と一対一に対応します。

コーディングを行う際には、あらかじめ定義されているニーモニック(nemonic)とフォーマット(次項で示す書式)に従ってください。

実行命令ステートメントは、アセンブラによって、それぞれ2～10バイトの機械語命令に変換されます。

2.　アセンブラ擬似命令ステートメント

アセンブラ擬似命令ステートメントは、アセンブラを制御するために使用します。

アセンブラ擬似命令による制御には、データの定義、データの確保、メモリ領域の確保などがあります。

このアセンブラ擬似命令の中では、データ定義のための擬似命令だけが機械語に変換されます。

3.　コメントステートメント

コメントステートメントは、プログラムを読みやすくし、プログラムのデバッグと保守を容易にするために使用します。

コメントステートメントは機械語には変換されません。

コメントの書き方には、次の2つの方法があります。

(a) ステートメントの先頭(第1カラム)にアスタリスク（*）を書くと、ステートメント全体がコメントになります。
これがコメントステートメントです。

(b) 実行命令ステートメント、アセンブラ擬似命令ステートメントのオペランドフィールドの後に、1文字以上のスペースを空けてアスタリスクを打ち、コメントを書きます。

例

```
      コメント(a)
  ┌─第1カラム
  ↓
＊10月10日にこのモジュールを更新←コメントステートメント…
                                  コメント行
jmp LABEL                ＊ LABEL へ分岐
    ↑                          ↑
  実行命令                  コメント(b)
```

1.1.3 ステートメントの書式

実行命令ステートメントおよび擬似命令ステートメントは、

- ラベルフィールド
- オペレーションフィールド
- オペランドフィールド
- コメントフィールド

の4つのフィールドから構成されています。

フィールドの構成

ラベル フィールド	オペレーション フィールド	オペランド フィールド	コメント フィールド

ステートメントの記述はフリーフォーマットです。
各フィールドの間は必ず1文字以上のスペースで区切ります。
ラベルの後には：(コロン)をつけます。

1.1 アセンブラプログラムの構成要素

例

LABEL :	move.w	d1,(a0)+	*コメント
ラベル フィールド	オペレーション フィールド	オペランド フィールド	コメント フィールド

1. ラベルフィールド

ラベルフィールドには、実行命令で参照する分岐先およびアセンブラ擬似命令で定義したシンボルを書きます。

●任意のカラムから記述して、コロン(:)で終わるシンボル

なお､アセンブラの予約語である次のシンボルは、ラベルとしては使えません。

●データレジスタ　D0～D7, d0～d7, R0～R7, r0～r7
●アドレスレジスタ　A0～A7, a0～a7, R8～R15, r8～r15
●その他のレジスタ　CCR, SR, SP, USP, SSP, PC
ccr, sr, sp, usp, ssp, pc
●実行命令
●アセンブラ擬似命令

2. オペレーションフィールド

オペレーションフィールドには、

●実行命令
●アセンブラ擬似命令
●マクロ呼び出し命令

のニーモニックを書きます。

処理するデータのサイズは、オペレーションのニーモニックの後に、ピリオド(.)に続けてデータサイズコードを併記して指定します。

● b　バイトデータ
● w　ワードデータ
● l　ロングワードデータ
● s　ショート(Bcc, BSR 命令用)

サイズを指定できるデータの処理を行う命令では、データのサイズを指定する必要があります。
指定を行わないと、デフォルトとしてワード(.w)がデータサイズとして想定されます。
データのサイズが決まっている命令の場合は、データサイズの指定は必要ありません。

例1

```
(1)  move. b    d0, d1    バイトデータの転送
(2)  move. w    d0, d1    ワードデータの転送
(3)  move__     d0, d1    ワードデータの転送
         ↑
         └──────デフォルト(.w)
(4)  move. l    d0, d1    ロングワードデータの転送
```

注 ： (2)、(3)は同じ処理を行います。

例2

```
lea      adr, a0    アドレスの転送 (ロングワード)
lea.l    adr, a0    アドレスの転送 (ロングワード)
    ↑
    └──────指定してもしなくてもよい
```

注 ： leaではデータサイズが決まっているので、指定の必要はありません。

3. オペランドフィールド

オペランドフィールドには、オペレーションフィールドの命令が必要とするオペランドを記入します。
命令によっては、オペランドが必要でないものがあります。
また複数のオペランドを必要とするものもあります。

4. コメントフィールド

コメントフィールドは、命令の処理の説明などのコメントを書くために使用します。

1.1.4 識別名

識別名には、ユーザーが自由に定義するシンボルと、アセンブラに対してすでに意味が決まっているキーワードの２つがあります。

シンボル

シンボルには次のものがあります。

ラベル

命令のロケーションアドレスを保持します。
一般に分岐先を指定する場合に使用します。

定　数

equ 擬似命令でシンボルに値が設定されると、以後このシンボルを定数として使用することができます。

変　数

set 擬似命令で値が設定されたシンボルのことです。
equ 擬似命令で定義された場合と違って、値の変更が可能です。

レジスタリスト

reg 擬似命令でレジスタ群を定義したシンボルです。

アスタリスク(＊)

現在のロケーションアドレスを保持します。

キーワード

キーワードには以下の２種類があります。

命令のニーモニック

各実行命令をニーモニックで表したものです。
各命令のニーモニックコードについては、「付録」を参照してください。

レジスタ名

レジスタ名をニーモニックコードで表したもので、以下のものがあります。

d0～d7、r0～r7 …………データレジスタ
a0～a7、r8～r15 ………アドレスレジスタ
ssp ………………………システムスタックポインタ
usp ………………………ユーザースタックポインタ
sr …………………………ステータスレジスタ
ccr ………………………コンディションコードレジスタ
pc …………………………プログラムカウンタ
a7、sp ……………………カレントスタックポインタ

なお、これらのニーモニックは大文字でも小文字でも表記可能です。

1.1.5 定　数

定数には、数値定数と文字定数があります。

数値定数

数値定数には、10 進数、16 進数、8 進数、2 進数の 4 種類があります。

10進数

10 進数は 0、1、2、…、9 の数字で表します。
先頭に特殊文字の指定がない数は 10 進数とみなします。

16進数

16 進数は 0、1、2、…、9、a、b、…、f で表し、その先頭にドル記号($)をつけて 16 進数であることを示します。

8進数

8 進数は 0、1、2…、7 の数字で表し、その先頭にアットマーク（@）をつけて 8 進数であることを示します。

2進数

2 進数は 0、1 の数字で表し、その先頭にパーセント記号(%)をつけて 2 進数であることを示します。

例 1

10 進数	4096
16 進数	$1000
8 進数	@10000
2 進数	%1000000000000

また見やすいようにセパレータとして'_'を使うことができます。

例 2

```
move   #%0100_0101_1100, d0
```

文字定数

文字定数はアポストロフィ(')で文字列を囲んで表します。
文字列の各文字は ASCII コードに変換されます。

例

dc. l	'AB'	ロングワードデータ$00004142 をメモリに確保
dc. w	'A'	ワードデータ$0041 をメモリに確保
dc. w	'A␣'	ワードデータ$4120 をメモリに確保

1.1.6 演算子

オペランドで使用する演算子には、単項演算子と二項演算子があります。

単項演算子

＋	この演算子に続く値が正であることを示します。
－	この演算子に続く値が負であることを示します。
.not.	論理 NOT
.high.	ロングワード(32 ビット)の下位ワードの上位バイトを分離し

	ます。
.low.	ロングワード(32 ビット)の下位ワードの下位バイトを分離します。
.highw.	ロングワード(32 ビット)の上位ワードを分離します。
.loww.	ロングワード(32 ビット)の下位ワードを分離します。

二項演算子

.mod.	剰余
.shr. (>>)	指定されたビット数だけ右へ論理シフト
.shl. (<<)	指定されたビット数だけ左へ論理シフト
.asr.	指定されたビット数だけ右へ算術シフト

算術シフトにおいては、最上位ビットは符号とみなされるのでこの内容はけして変わりません。
これに対して論理シフトは、データを数ではなく論理的な0と1の集まりとみなしているのですから、符号というものはありえません。
右へシフトした場合でも、左へシフトした場合でも、空いたビット位置には常に0が入ります。

例

```
move   #(100. mod. 17), d0
  100÷17 の剰余値 15 を d0 に設定
move   #(%11000000. shr. 5), d0
  %11000000 を 5 ビット右へ論理シフトした値%00000110 を d0 に設定
move   #(%00000011. shl. 5), d0
  %00000011 を 5 ビット左へ論理シフトした値%01100000 を d0 に設定
move. l   #($ffff0000. asr. 5), d0
  $FFFF0000 を 5 ビット右へ算術シフトした値$FFFFF800 を d0 に設定
```

＋　加算
－　減算
＊　乗算
／　除算

.eq. (＝)　オペランドが互いに等しいとき真（－1)を返します。
.ne. (<>)　オペランドが互いに等しくないとき真（－1)を返します。

.lt. (<)　左オペランドが右オペランドより小さいとき真(－1)を返します。

.le. (<＝)　左オペランドが右オペランドより小さいか等しいとき真(－1)を返します。

.gt. (>)　左オペランドが右オペランドより大きいとき真(－1)を返します。

.ge. (>＝)　左オペランドが右オペランドより大きいか等しいとき真(－1)を返します。

.slt.　符号つき lt

.sle.　符号つき le

.sgt.　符号つき gt

.sge.　符号つき ge

.and.　論理 AND。
両オペランドの論理積を返します。

.or.　論理 OR。
オペランドの論理和を返します。

. xor.　排他的 OR。
両オペランドの排他的論理和を返します。

1.1　アセンブラプログラムの構成要素

演算子の優先順位は次の通りです。

①	(　)で囲まれた式(内側をより優先)	
②	単項演算子	＋, －, .not., .high. , .low., highw., .loww.
③	乗算、除算、剰余、シフト	＊, ／, .mod., .shr., .shl., .asr.
④	加算、減算	＋, －
⑤	大小比較演算子(.eq.など)	.eq., .ne., .lt., .le., .gt., .ge., .slt., .sle., .sgt., .sge.
⑥	論理 AND	.and.
⑦	論理 OR、排他的 OR	.or., . xor.

1.2 アドレス形式

アセンブラの命令の実行にあたっては、操作対象となるオペランド実効アドレスの指定が必要になります。
この実効アドレスの指定方法を、アドレス形式といいます。
ここでは、アドレス形式を以下の項目ごとに、例を用いて説明します。

- ●レジスタ直接アドレッシング (1. 2. 1)
- ●アドレスレジスタ間接アドレッシング (1. 2. 2)
- ●絶対アドレッシング (1. 2. 3)
- ●プログラムカウンタ相対アドレッシング (1. 2. 4)
- ●イミディエートデータアドレッシング (1. 2. 5)

1.2.1 レジスタ直接アドレッシング

操作の対象がレジスタで、指定したレジスタの内容に対して命令が直接に実行されるモードです。
このモードには、レジスタの種類により以下のものがあります。

- ●データレジスタ直接形式
- ●アドレスレジスタ直接形式

データレジスタ直接形式
操作の対象となるデータは指定したデータレジスタの内容です。

表記法　dn, rn (n は 0～7)

例

```
clr.l   d0    d0をゼロクリア
```

アドレスレジスタ直接形式

操作の対象となるデータは指定したアドレスレジスタの内容です。

表記法　an (n は 0～7)　rn (n は 8～15)

例

```
add.l  a1,a2
```

a1 のロングワードを a2 のロングワードに加算して、結果の値を a2 へ格納します。

1.2.2 アドレスレジスタ間接アドレッシング

操作の対象は、指定したアドレスレジスタの内容でポイントされるメモリ内のデータです。

また、実効アドレスの決定にあたってアドレス修飾が行われる場合もあります。

このモードのアドレス指定形式には、次のものがあります。

- ●アドレスレジスタ間接形式
- ●ポストインクリメントアドレスレジスタ間接形式
- ●プレデクリメントアドレスレジスタ間接形式
- ●ディスプレースメントつきアドレスレジスタ間接形式
- ●インデックスつきアドレスレジスタ間接形式

アドレスレジスタ間接形式

オペランドのアドレスは、レジスタフィールドで記述したアドレスレジスタの内容です。

表記法　(an)

例

```
move　#1, (a1)
  値1をa1の内容がポイントするアドレスのメモリへ転送します。

sub. w　(a1), d1
  d1から、a1の内容がポイントするアドレスのメモリの値を減算し、
  その結果をd1に格納します。
```

ポストインクリメントアドレスレジスタ間接形式

オペランドのアドレスはレジスタフィールドで記述したアドレスレジスタの内容です。

オペランドのアドレスが参照された後に、オペランドのサイズがバイト(.b)であれば1を加算します。

またワード(.w)であれば2が、またロングワード(.l)であれば4が加算されます。

表記法　(an)+

例

```
move. b　(a2)+, d1
  a2の内容がポイントするアドレスのメモリの内容を1バイトでd1へ
  転送します。
  転送後a2は1が加算されます。

move. w　(a2)+, d1
  a2の内容がポイントするアドレスの内容を1ワードでd1へ転送しま
  す。
  転送後a2は2が加算されます。
```

プレデクリメントアドレスレジスタ間接形式

オペランドのアドレスはレジスタフィールドで記述したアドレスレジスタの内容です。

オペランドのアドレスが参照される前に、オペランドのサイズがバイト(.b)

であれば1を減算します。
ワード(.w)であれば2を、またロングワード(.l)であれば4を減算します。

表記法　−(an)

例

clr −(a1)
a1から2を減算し、その結果がポイントするアドレスの内容を1ワードゼロクリアします。

cmp. w −(a1), d1
a1から2を減算し、その結果がポイントするアドレスの内容とd1の内容をワード単位で比較します。

ディスプレースメントつきアドレスレジスタ間接形式
オペランドのアドレスは、アドレスレジスタの内容と符号ビット拡張のディスプレースメントとの加算値となります。

表記法　d (an)

例

DISP equ 3
DISPは次の命令のために3に定義されます。

clr. b DISP (a1)
DISPの値とa1の内容を加算した値をアドレスとして、そのアドレスの内容を1バイトクリアします。

move #4, 10 (a2)
10(値)をa2の内容に加算し、その値によって指定される値をアドレスとするワードの領域に4を転送します。

インデックスつきアドレスレジスタ間接形式

オペランドのアドレスは、アドレスレジスタの内容、符号ビット拡張のディスプレースメント、およびインデックスレジスタ(インデックスレジスタとして、アドレスレジスタ、データレジスタのいずれも使用することができます）の内容の加算値です。

表記法　d (an, rn. w)

インデックスレジスタとしてrnの下位ワードが符号ビットを拡張されて使用されます。

d (an, rn. l)

インデックスレジスタとしてrn全体が使われます

例1

```
add  DISP (a1, d1), d3
```

a1の内容、インデックスレジスタd1の下位ワードの内容およびディスプレースメントDISPの和をアドレスとするワードの内容を、d3の下位ワードへ加え、結果をd3に格納します。

例2

```
move. l  d3, $20(a2, a3.l)
```

a2の内容、インデックスレジスタa3およびディスプレースメント値$20の和をアドレスとするロングワードの領域へd3全体の値を転送します。

1.2.3 絶対アドレッシング

操作対象であるデータの格納アドレスを、絶対アドレスで直接指定するモードです。

このモードには、アドレスのデータ長によって次の2種類があります。

- ●絶対ショートアドレス形式
- ●絶対ロングアドレス形式

絶対ショートアドレス形式

オペランドで指定できるアドレスは1ワード、すなわち16ビットですが、アドレス指定には32ビット必要となるために、参照前に、32ビットに符号つきで拡張されます。
このため有効なアドレスの範囲は0から$7fff、さらに$ffff8000から$ffffffffになります。

例

```
jmp $600
  $600 番地へジャンプ
```

絶対ロングアドレス形式

オペランドのアドレスは32ビット値です。

例

```
jmp $14000
  $14000 番地へジャンプ
```

1.2.4 プログラムカウンタ相対アドレッシング

操作対象となるデータの格納アドレスは、プログラムカウンタの現在値からの相対アドレスで表されます。
このモードには、次の2種類があります。

- ●ディスプレースメントつきプログラムカウンタ相対形式
- ●インデックスつきプログラムカウンタ相対形式

ディスプレースメントつきプログラムカウンタ相対形式

オペランドのアドレスは、プログラムカウンタ中のアドレス値と、符号ビット拡張後のディスプレースメントの和です。

表記法　＜式＞(pc)

例

```
jmp TAG (pc)
  TAG＋プログラムカウンタが示すアドレスへ分岐
```

インデックスつきプログラムカウンタ相対形式

オペランドのアドレスは、プログラムカウンタの内容とインデックスレジスタの(アドレスレジスタまたはデータレジスタを使用します)内容の加算値です。

表記法　＜式＞(pc, rn.w)
　　インデックスレジスタとしてrnの下位ワードが符号ビットを拡張されて使用されます。
　　＜式＞(pc, rn.l)
　　インデックスレジスタとしてrn全体が使われます。

例

```
jmp  (pc, a2. w)
  プログラムカウンタのアドレス値とa2の下位ワードを符号拡張されたアドレス値の和が示すアドレスへの分岐。
```

1.2.5 イミディエートデータアドレッシング

操作対象となるデータを直接指定するモードであり、次の3種類があります。

- イミディエートデータ形式
- クイックイミディエート形式
- SR/CCR 形式

イミディエートデータ形式

"#"の後に値または式を書くことで、値そのものをオペランドとします。
"#"はイミディエート文字と呼びます。

1.2　アドレス形式

表記法　#<式>

例

move #3, d1
　値3をd1のワード(下位)へ転送します。

sub. w #3,d1
　d1のワード(下位)から値3を減算し、結果をd1のワード(下位)に格納します。

クイックイミディエート形式

イミディエートデータ形式とは異なり、イミディエート値が拡張ワード内ではなく、オペレーションワード内に存在するのでオブジェクトも短く、高速処理が可能となります。
この形式がサポートされている命令は、次の3種類です。

- moveq　　1バイトデータの高速転送
- addq　　1〜8のインクリメント命令
- subq　　1〜8のデクリメント命令

例

addq. b　#3, (a1)
　a1がポイントするメモリ(バイト)の内容に3を加算します。

moveq. l　#3,d1
　d1に3をロードする。
　ただし転送されるサイズはロングワードに符号拡張されます。

1.2 アドレス形式

SR/CCR 形式

対象となるデータは、ステータスレジスタ (sr) またはコンディションコードレジスタ(ccr ステータスレジスタの下位バイトです) の内容です。
なお、SR を扱う命令の大部分は特権命令です。
この形式の命令セットは以下の通りです。

andi	to	sr	andi	to ccr
eori	to	sr	eori	to ccr
ori	to	sr	ori	to ccr
move	to	sr	move	to ccr
move	from	sr		

例

```
move. w   $1040, sr
```

アドレス$1040 のワードデータが sr に設定されます。

1.3 リロケータブルなプログラム作成

X68000で作成されるプログラムはリロケータブル(再配置可能)な構造となっています。

リロケータブルなプログラムとは、主記憶上のどの領域に置かれても実行可能なプログラムのことです。

リロケータブルなプログラムにおいては、プログラム内部のデータの参照はプログラム内相対のアドレスによって、モジュールおよびサブルーチンの呼び出しは、プログラム内相対のディスプレースメントにもとづいて行われるため、実アドレス、すなわちプログラムのロードアドレスを意識しなくても済むようになっています。

これらデータのアドレスや、モジュールおよびサブルーチンの分岐先アドレスは、実行時に実アドレスに変換されます。

本節では、リロケータブルなプログラムを作成するときに知っておくべきことと、これに関連する擬似命令について述べます。

1.3.1 セクション

リロケータブルなプログラムを作成するにあたっては、ソースプログラムをセクションと呼ばれるまとまりに分けますが、コーディング段階では、メモリ上のどの領域にロードするのか決まっていないセクションをリロケータブルなセクションと呼びます。

すなわち、別々にコーディングし、別々にアセンブルしたプログラムであっても、実行命令の領域や変数の領域を同一番号のセクションとしてソースプログラムで指定しておけば、実行命令の領域、変数の領域としてそれぞれ1つにまとめることができます。

セクションの種類と、これを定義する擬似命令を以下に示します。

テキストセクション……………text 擬似命令
データセクション…………………data 擬似命令
ブロックストレージセクション…bss 擬似命令
スタックセクション……………stack 擬似命令

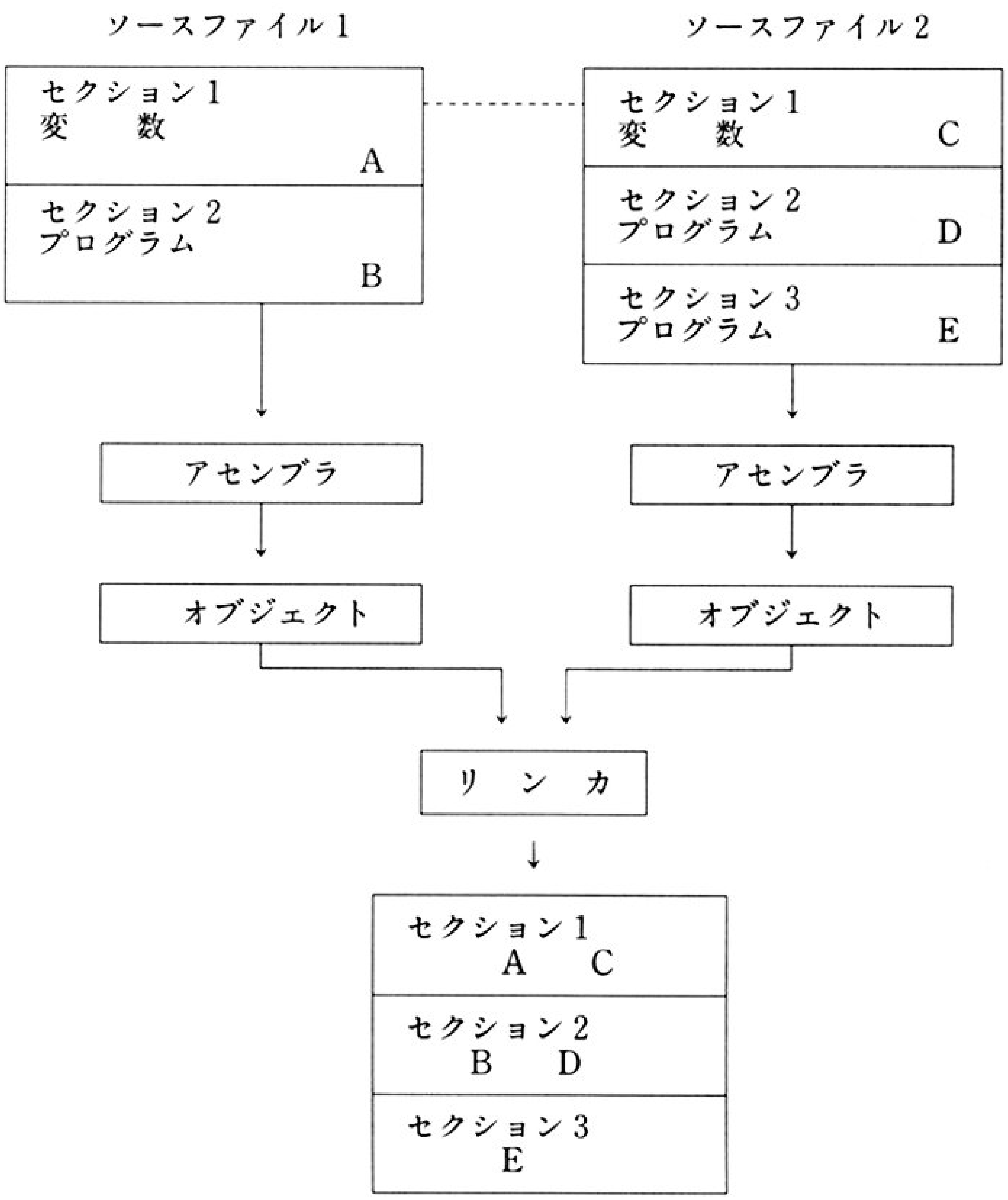
各セクションには、それぞれ独立したロケーションカウンタが割りあてられます。

プログラムの中で番号の違うセクションが新しく定義されると、ロケーションカウンタは0となります。
以前と同じ番号のセクションが再び定義されると、ロケーションカウンタの値は以前の定義部分の続きの値となります。
なおロケーションカウンタの値は、各セクションの先頭から相対アドレスなので、メモリのロードアドレス(実アドレス)とは関係ありません。

1.3.2 共通データエリア

複数のプログラムから共通に使用される変数やデータは、共通データエリアとして宣言しなくてはなりません。
共通データエリアは、それを参照するプログラムにある必要はなく、リンカによって共有する1つの領域が割りあてられます。

共通データエリア

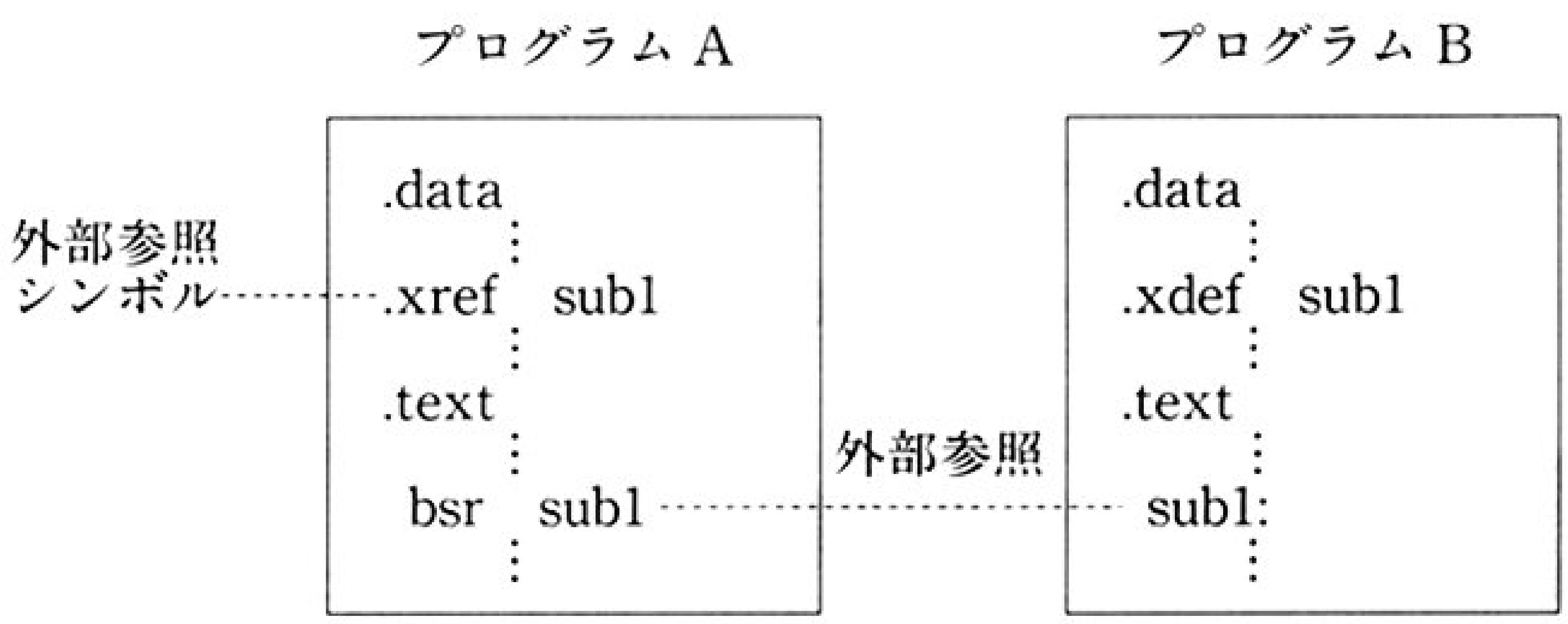


図ではプログラムを2つのモジュール(ソースファイル1)と(ソースファイル2) に分割して作成し、リンカで1つのプログラムにまとめています。
このとき2つのプログラムの各セクションは図のようになります。

1.3.3 外部参照

それぞれ別々にアセンブルしたオブジェクトモジュールを結合して動作させる場合、あるプログラムから他のプログラム内のシンボルを参照することがあります。
このように他のプログラムのシンボルを参照することを外部参照といいます。
逆に、他のプログラムから参照されるシンボルを外部定義といいます。

外部参照



外部参照の対象となるシンボルのアドレスの決定は、リンカが行います。
すなわちリンカは、複数個のオブジェクトモジュールの結合にあたって、これらを一定の順序で配置(通常はオブジェクトモジュール名の入力順です)し、各モジュールにアドレスの再割りあてを行うとともに、各シンボルのアドレスもこれにともなって更新を行い、外部参照シンボルの決定を行います(むろんここでいうアドレスとは、最終的には1個となる実行可能プログラムの先頭からの相対アドレスであり、メモリ上のロードアドレスすなわち実アドレスとは関係ありません)。

外部参照および外部定義の宣言は、xdef、xref 擬似命令によって行います。
必要ならユーザーは、ソースプログラム中に xdef、xref 擬似命令を定義しなければなりません。
アセンブラはこれらの擬似命令を検出すると、リンカにわたす情報を作成し、リンカはこれにもとづいて外部参照シンボルの解決を行います。

1.4 アセンブラ擬似命令

アセンブラ擬似命令は、実行命令とは異なり、一部(.dc, .dcb など) を除けばオブジェクトコードに変換されることはありません。
この命令は、アセンブリ作業などに関する情報をアセンブラに与えます。
擬似命令を機能別に分けると、次のようになります。

本節では、アセンブラ擬似命令を上記の項目ごとに例を用いて説明します。

1.4.1 アセンブラ制御

セクションやロケーションの指定などのように、オブジェクトモジュールを作成するにあたって必要となる情報やデータをアセンブラに指示します。
この擬似命令には以下のものがあります。

- .text テキストセクションの指定
- .data データセクションの指定
- .bss ブロックトレースセクション(bss)の指定
- .stack スタックセクションの指定
- .offset オフセットの指定
- .include ソースコードの挿入
- .comm コモンエリアの指定
- .end プログラムの終了指定
- .org ロケーションカウンタの指定
- .comment コメント行の指定
- .fail エラーの生成

.text テキストセクションの指定

書式

.text

機能

テキストセクションの宣言を行います。

解説

プログラムのテキストセクションプログラムコードの定義部の開始を宣言します。

例

```
.text

        lea     JP_TBL,a0
        lsl.l   #3,d0
        jmp     2(a0,d0.l)
```

.data データセクションの指定

書式　.data

機能　データセクションの宣言を行います。

解説　プログラムのデータセクション(初期値つきデータ部)の開始を宣言します。

例

```
.data

*       ------------
*       message area
*       ------------

MSG_0:  .dc.b   'fatal error'
MSG_1:  .dc.b   'file not found'
MSG_2:  .dc.b   'parity error'
```

.bss　ブロックストレージセクションの指定

書　式　.bss

機　能　ブロックストレージセクション（bss）の宣言を行います。

解　説　プログラムのブロックストレージセクション(初期値なしデータ部)の開始を宣言します。

例

```
.bss

*       --------------------
*       completion code area
*       --------------------

Code_0: .ds.w   1
Code_1: .ds.w   1
Code_2: .ds.w   1
```

.stack スタックセクションの指定

書　式　.stack

機　能　スタックセクションの宣言を行います。

解　説　プログラムのスタックセクションの開始を宣言します。

例

```
.stack

*        -------------
*        stack section
*        -------------

stk:     .ds.w   240
```

.offset オフセットの指定

書　式　.offset　＜式＞

機　能　オフセット表を定義します。

解　説　.offset 擬似命令は、.ds 擬似命令によって定義するオフセット表の開始アドレスを指定します。
このオフセット表は一種の擬似セクションであり、リンカにはわたされません。
オフセット表で定義されたシンボルは、モジュール内部に確保されます。
オフセット表で使用可能な擬似命令には以下のものがあります。

.ds
.equ
.set
.reg
.xdef
.xref

なお、オブジェクトコードに変換される実行命令、または擬似命令は指定できません。
＜式＞の値はオフセット表の開始アドレスです。
この式は絶対値でなければならず、前方参照、未定義値、外部参照などを含んではなりません。
.offset 擬似命令は、以下の擬似命令のいずれかにより終了します。

.text
.data
.bss
.stack
.end

例

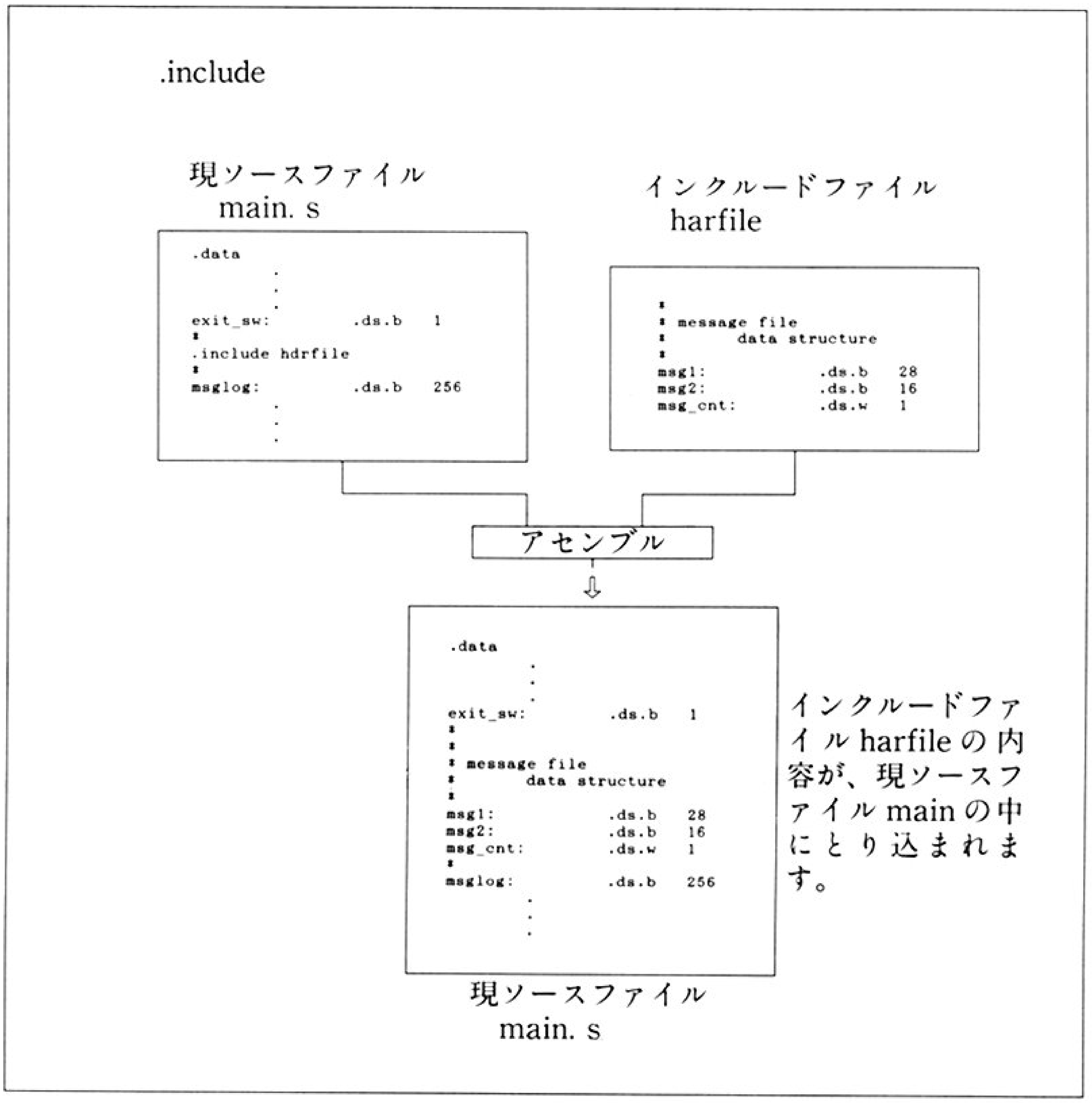
```
.offset 600

SATA01: .ds.w   1
SATA02: .ds.w   1
SATA03: .ds.w   1
SATA04: .ds.w   1
```

.include ソースコードの挿入

書式	.include ＜ファイル名＞
機能	ソースコードをソースファイルに挿入します。
解説	アセンブル中に、現ソースファイルにインクルードファイルからソースコードを挿入します。

例



.comm　コモンエリアの指定

書　式　.comm　<ラベル>, <式>

機　能　コモンエリアの指定を行います。

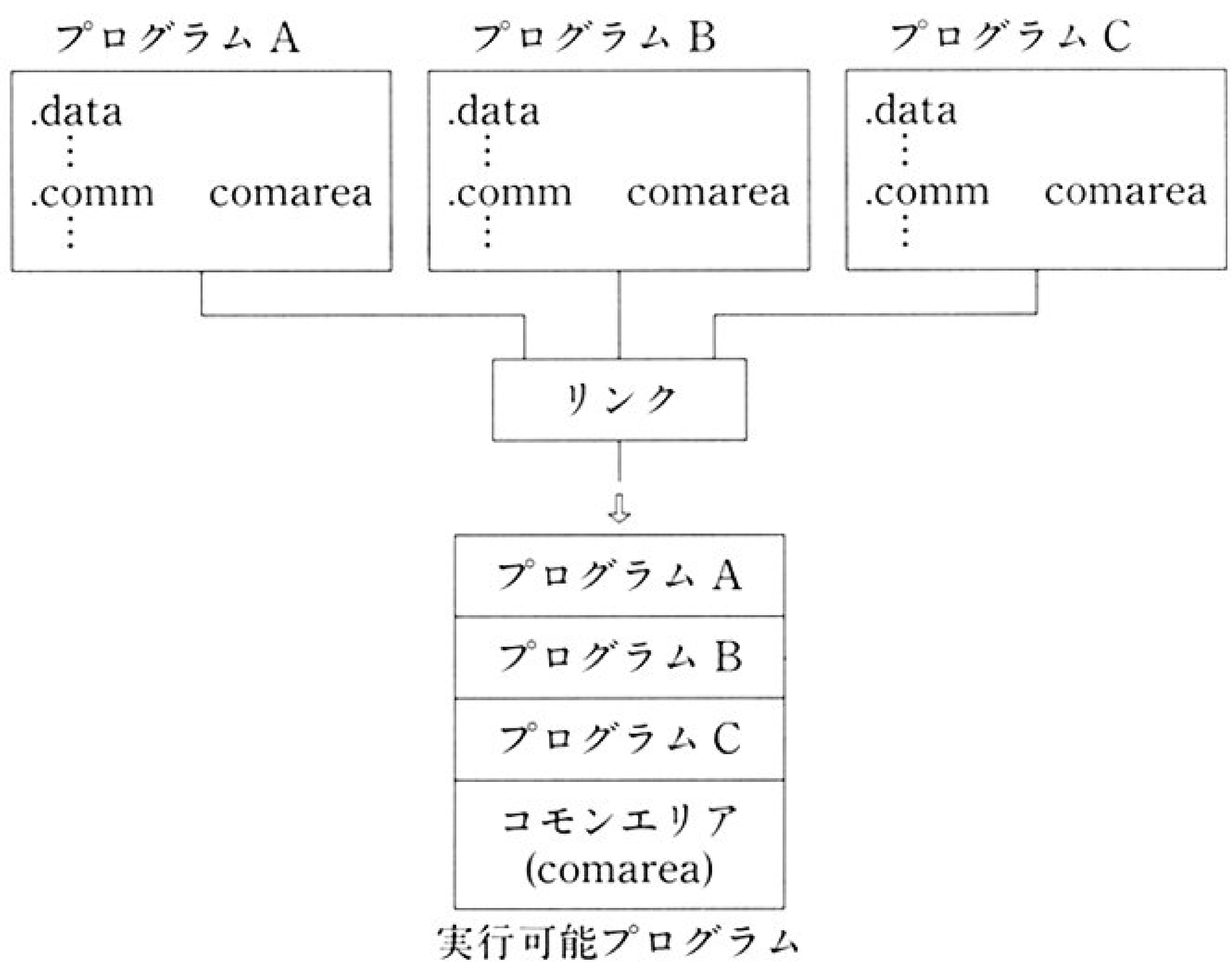
解　説　コモンエリアのラベルとサイズを指定します。

コモンエリアは別々にアセンブルされたプログラム間で共有することができます。

リンカは、同じラベルをもつすべてのコモンエリアを同一アドレスに割りつけます。

同じラベルをもつコモンエリアのサイズが異なる場合には、その中での最大のサイズが割りあてられます。

例



.end プログラムの終了指定

書式 .end <ラベル>

機能 プログラムの終了を指定します。
またラベルを指定すると、プログラムの実行開始アドレスとなります。

解説 .end 擬似命令はアセンブラに、ソースプログラムの終了を通知します。
したがって後続のソースコードはすべて無視されます。

例

```
        move.l  par1+6(sp),a0
        bsr     super_on
        clr.l   d1
        move.b  (a0),d1
.end
        bsr     super_off  ┐
        bra     return_d1  ┘ この2行は無視されます。
```

注：複数のオブジェクトをリンクする場合には1つのオブジェクトのみにラベル指定がされている .end 擬似命令を書いてください。

.org　ロケーションカウンタの指定

書　式　.org　<式>

機　能　ロケーションカウンタの指定

解　説　現在のセクションのロケーションカウンタを<式>の値にします。

例

```
.org   $1000
```

.comment　コメント行の指定

書式　.comment　＜文字列＞

機能　コメント行の指定

解説　＜文字列＞で指定された文字列が再び現れる行まで、コメント行とします。
指定された文字列が現れた行は、文字列が行の先頭に現れてもコメント行となります。
.comment のネストはできません。

例

```
          .commemt          note

          moveq    #10,d0          ─┐
          ここはコメント行です.       │
          movea.l  (sp)+,a0          ├── コメント行
note      lea      16(sp),sp       ─┘
          tst.l    d0              ──────── アセンブラの命令として有効
```

.fail　エラーの生成

書　式　.fail　<式>

機　能　エラーの生成

解　説　<式>の結果が真ならば、アセンブラ時にエラーを発生します。

例

```
.fail    stacksiz.gt.$1000
```

シンボル stacksiz が$1000 より大きい値なら、エラーを発生し、アセンブルを中断します。

1.4.2 外部名の指定

同時にリンクされるプログラム間で参照するシンボルを定義します。
この擬似命令には以下のものがあります。

- .globl, .global　　グローバルシンボルの宣言
- .xdef, .public, .entry　　外部定義名の宣言
- .xref, .extrn, .exteral　　外部参照名の宣言

.globl, .global　グローバルシンボルの宣言

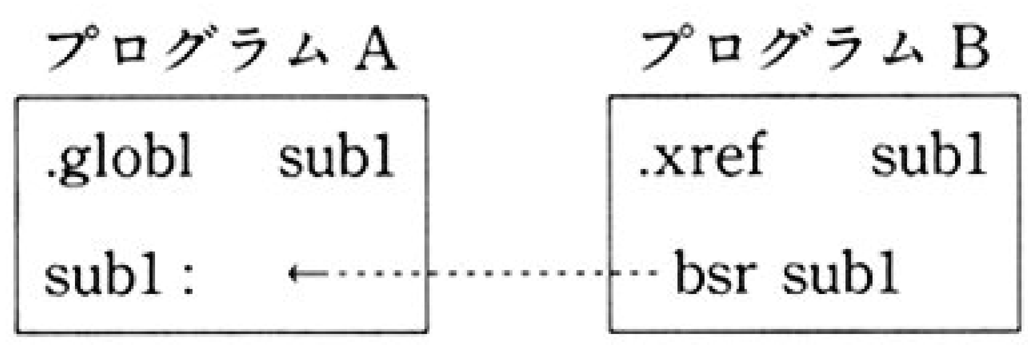
書　　式　.globl　＜ラベル＞ [,＜ラベル＞・・・・]
.global　＜ラベル＞ [,＜ラベル＞・・・・]

機　　能　＜ラベル＞を外部名として定義します。

解　　説　＜ラベル＞を外部名として定義します。
該当プログラムが他のプログラムとリンクされると、この＜ラベル＞は他のプログラムから参照することができます。
逆に指定された＜ラベル＞がソースコード中に定義されなかった場合、リンカはこの＜ラベル＞が他のプログラム中に定義されているものとみなします。

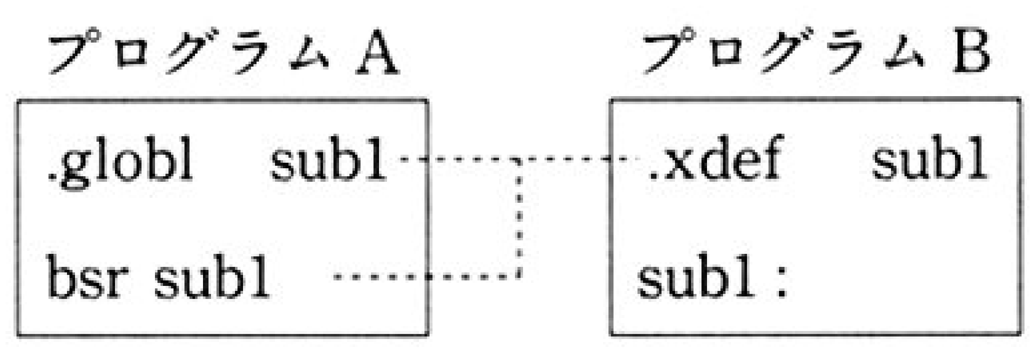
globl

＜ラベル＞が同一プログラム内で定義されている場合



ラベルsub1はプログラムBより参照可能となります。

＜ラベル＞が同一プログラム内で定義されていない場合



ラベルsub1は他のプログラムにあるものとみなされます。

.xdef, .public, .entry　外部定義名の宣言

書式

.xdef　＜ラベル＞ [,＜ラベル＞・・・・]

.public　＜ラベル＞ [,＜ラベル＞・・・・]

.entry　＜ラベル＞ [,＜ラベル＞・・・・]

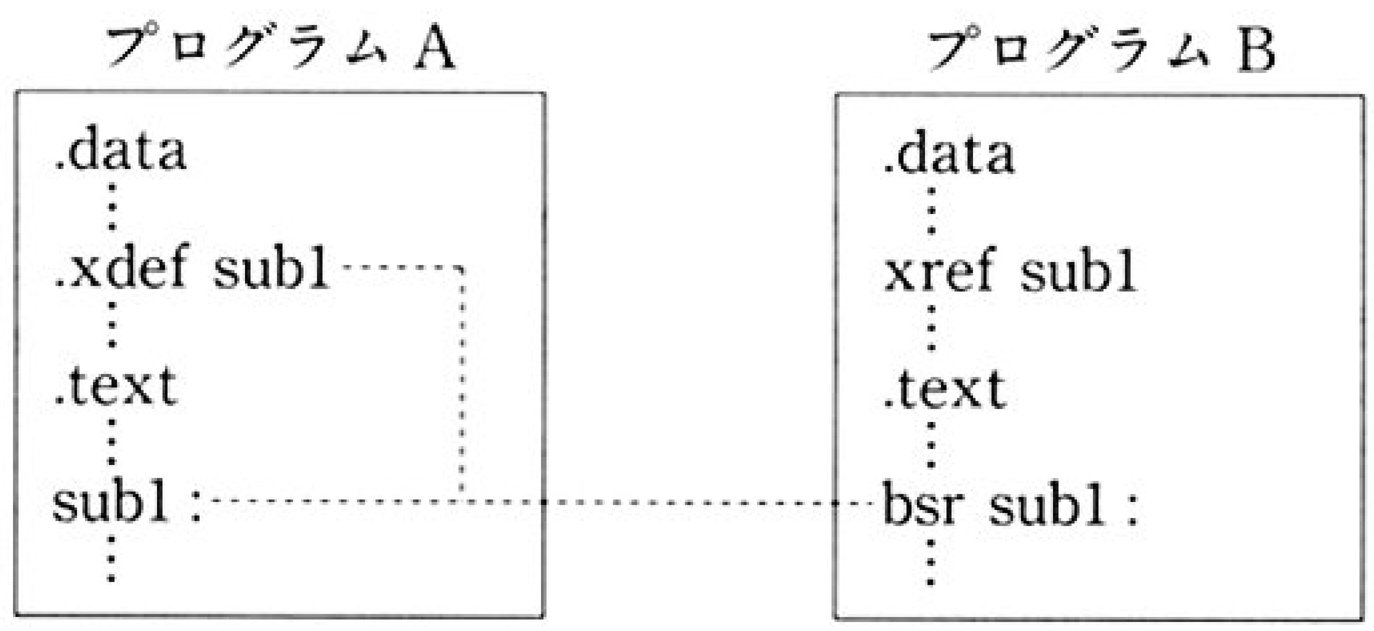
機能

外部から参照可能なシンボルとして定義します。

解説

.xdef, .public, .entry 擬似命令は、この擬似命令が記述されたモジュールと共にリンクされる他のモジュールから参照されるシンボルを定義します。

例



上記の例では、プログラムBから呼び出されるサブルーチン sub1 を、.xdef 擬似命令で定義しています。

.xref, .extrn, .exteral　外部参照名の宣言

書式

.xref　＜ラベル＞［,＜ラベル＞・・・・］
.extrn　＜ラベル＞［,＜ラベル＞・・・・］
.exteral　＜ラベル＞［,＜ラベル＞・・・・］

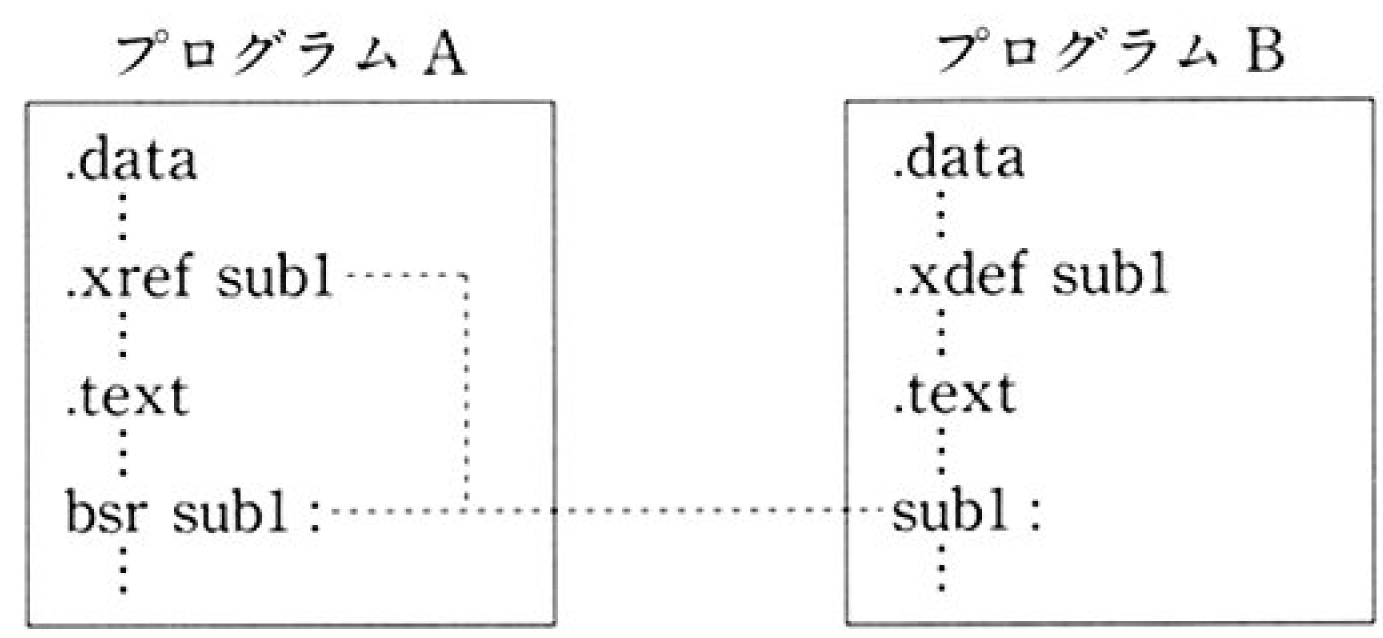
機能

外部で定義されたシンボルを参照することを宣言します。

解説

.xref, .extrn, .exteral 擬似命令は、他のモジュールで定義されたシンボルを、この擬似命令が記述されたモジュールから参照することを宣言します。

例



上記の例では、プログラムB内のサブルーチン sub1 を、プログラムAから呼び出すことを.xref 擬似命令で宣言しています。

1.4.3 シンボル値の定義

シンボルに設定する値を定義します。
この擬似命令には以下のものがあります。

- equu　不変シンボル値の定義
- set　可変シンボル値の定義
- reg　レジスタリストの定義

equ　不変シンボル値の定義

書式　<シンボル>　equ　<式>

機能　不変シンボルの値を定義します。

解説　equ擬似命令は、オペランドフィールドの<式>の値をラベルフィールドのシンボルに割りあてます。

ラベルフィールド、オペランドフィールドはともに必要です。

なお、シンボルは固有のものでなければなりません。

一度シンボルを定義すると、そのシンボルはプログラム中の他の箇所で再定義することはできません。

また、式には未定義のシンボルや、この式の後に定義されるシンボルを含んではなりません。

例

```
chknum  equ     6
chksum  equ     2
mulchk  equ     chknum*chksum
addchk  equ     chknum+chksum
```

set、= 不変シンボル値の定義

書　式　＜シンボル＞　set　＜式＞
＜シンボル＞　=　＜式＞

機　能　シンボルに一時的な値を割りあてます。

解　説　set 擬似命令は、オペランドフィールドの＜式＞の値をラベルフィールドのシンボルに割りあてます。
equ 擬似命令と異なるのは、set 擬似命令で定義したシンボルの値を、同じプログラム中の別の set 擬似命令で再定義できることです。
再定義は 2 回以上可能であり、また前の定義を参照することができます。
ラベルフィールド、オペランドフィールドはともに不可欠です。
オペランドフィールドには、未定義の式や、set 擬似命令が出現した時点でまだ定義されていないシンボルを含んではなりません。

例

```
msglen  set     28
blklen  set     4
msgblk  set     msglen/blklen
           .
           .
           .
msgcnt=0
```

reg　レジスタリストの定義

書式　<ラベル>　reg　<レジスタリスト>

機能　レジスタリストを定義します。

解説　reg擬似命令は、movem命令で使用するレジスタリストの値をラベルに割りあてます。
以後、レジスタリストの内容は、ラベルを指定するだけで一意に参照することができるようになります。
レジスタリストのフォーマットは以下の通りです。
レジスタ1［−レジスタ2］［／レジスタ3［−レジスタ4］］

例

```
.data
         .
         .
         .
reglist          reg      d0-d7/a0-a7
         .
         .
         .
.text
                 movem    reglist,(sp)+
```

レジスタリストで定義されたレジスタ群をメモリ領域へ退避します。

1.4.4 マクロ制御

一連の命令を1つの名前で参照できるようにすることをマクロ定義といいますが、本節で扱う擬似命令はこのマクロ定義に関する情報をアセンブラに通知します。
この擬似命令には以下のものがあります。

- .macro　　マクロ定義の開始
- .local　　マクロ定義ブロック内の局所的シンボルの定義
- .endm　　マクロ定義の終了
- .exitm　　マクロ展開の打ち切り

.macro　マクロ定義の開始

書　式　<ラベル>　.macro［<パラメータリスト>］

機　能　マクロ定義を開始します。

解　説　.macro擬似命令はマクロ開始文であり、ラベルはマクロが呼ばれるときのニーモニックです。

.macro擬似命令から.endm擬似命令までの間の記述がマクロとして定義されます。

<パラメータリスト>はマクロに与えられる引き数です。

例

```
macsub  .macro   param
        .local   final
        move.l   a0,save1
        move.l   a1,save2
        cmp      param.a1
        bne      final
        rts
final:  trap     #0
        .endm
```

.local マクロ定義ブロック内の局所的シンボルの定義

書式 .local <シンボル> [,<シンボル>・・・・]

機能 マクロ定義ブロック内の局所的シンボルの定義を行います。

解説 .local擬似命令は、マクロ定義ブロック内だけで使用するシンボルを定義します。
この擬似命令はマクロ定義ブロック内でのみ使用可能です。

例

```
macsub  .macro
        .local   final
        move.l   a0,save1
        move.l   a1,save2
        cmp      a0.a1
        bne      final
        rts
final:  trap     #0
        .endm
```

finalはマクロ定義ブロック内のみで使用される局所的シンボルです。

.endm　マクロ定義の終了

書　式　endm

機　能　マクロ定義を終了します。

解　説　マクロ定義を終了します。

例

```
macsub  .macro
        .local  final
        move.l  a0,save1
        move.l  a1,save2
        cmp     a0,a1
        bne     final
        rts
final:  trap    #0
        .endm
```

.exitm マクロ展開の打ち切り

書式　.exitm

機能　マクロ展開を途中で打ち切ります。

解説　.exitm擬似命令は、マクロ定義されたソース文の展開を途中で終了させます。これは条件つきアセンブリ擬似命令とともに用いられ、ある条件により残りのマクロ展開が不要、または不適切なものになった場合、.endm擬似命令までのマクロ展開を抑制することができます。

例

```
test    = 1
macsub  .macro
        .local  final
        move.l  a0,save1
        move.l  a1,save2
        cmp     a0.a1
        bne     final
        .if test
        .exitm
        .endc
final:  trap    #0
        .endm
```

1.4.5 データ定義・領域確保

データ領域の確保に関する情報をアセンブラに通知します。
この擬似命令には以下のものがあります。

- .dc　　定数データの定義
- .dcb　　定数ブロックの定義
- .ds　　メモリ領域の確保
- .even　　偶数バンダリの調整

.dc 定数の定義

書　　式　.dc[.<サイズ>] <式> [,<式>,・・・・]

機　　能　定数を定義します。

解　　説　.dc 擬似命令は、メモリ内に一個以上の定数を定義します。
複数の<式>を指定する場合は、各<式>をカンマ(,)で区切ります。
各<式>には以下のものを指定することができます。

1.　数値定数

- シンボル、または数値が割りあてられる式
- 10 進数
- 16 進数
- 8 進数
- 2 進数

2.　文字定数

ASCII コードで表される定数

ASCII コードの値を指定するときは、引用符 (') で文字を囲まなければなりません。
定数は、バイト、ワード、またはロングワードで指定することができます。
その場合、<サイズ>にそれぞれ、b、w、l を指定します。

.dc 擬似命令の表記法と規則は次の通りです。

.dc.b

バイト単位で定数を定義します。
奇数個のバイトを指定すると、次のステートメントが奇数番地から始まる場合がありますので注意してください。
.dc.b 擬似命令の、次の命令のワードバンダリ調整は原則として行いません。

.dc.w

ワード単位で定数を定義します。
奇数個のバイトを指定した場合、バイト数が偶数個になるように、最後のワードの上位に 0 を設定します。
.dc.w 擬似命令で奇数バイトの指定ができるのは、文字定数のときだけです。
この命令が奇数番地から始まる場合は、先頭に 1 バイト 0 が入りワードバンダリ調整します。

.dc.l

ロングワード単位で定数を定義します。

指定バイト数が4の整数倍に満たないときには、最後のロングワードの上位に0を設定します。

この命令が奇数番地から始まる場合は、先頭に1バイト0が入りワードバンダリ調整します。

.dc擬似命令の書式の例を以下に示します。

例

```
.dc.b   'error'
.dc.w   10
.dc.w   4
.dc.w   8

.dc.l   10
.dc.l   4
.dc.l   8
```

.dcb　定数ブロックの定義

書　　式

.dcb［.＜サイズ＞］＜長さ＞，＜式＞

機　　能

定数ブロックを定義します。

解　　説

.dcb 擬似命令は、＜サイズ＞に従って1ブロックのバイト(b)、ワード(w)、ロングワード (l) を割りあてます。

ブロック長は未定義参照、前方参照および外部参照を含まない絶対値をもつ＜長さ＞で決まります。

メモリ領域は＜式＞で指定したデータで満たされます。

例

```
.bcb     2,0
.bcb.b   2,0
.bcb.w   1,0
.bcb.l   1,0
```

.ds　メモリ領域の確保

書　式　.ds［.＜サイズ＞］＜長さ＞

機　能　メモリ領域を確保します。

解　説　.ds擬似命令は、メモリ領域を確保します。
確保されたメモリ領域の内容は初期化されません。
領域の確保は、バイト、ワード、ロングワード単位で行うことができます。
その場合、＜サイズ＞にそれぞれ、b、w、lを指定します。
＜長さ＞は、.ds擬似命令で確保するバイト、ワード、またはロングワードの数を指定します。

例

```
.ds      1
.ds.b    3
.ds.w    2
.ds.l    2
```

.even 偶数バンダリの調整

書式 .even

機能 偶数バンダリの調整を行います。

解説 .even 擬似命令は、次の命令またはデータフィールドを偶数番地に割りあてます。
たとえば、ロケーションカウンタが奇数となったときに、.even 擬似命令が指定されると、ロケーションカウンタは1つインクリメントされます。

例

```
00000000 01      flag1   .dc.b   1
00000001 00      flag2   .dc.b   0
00000002 00      flag3   .dc.b   0
00000003 00      .even
00000004 0001    msgcnt  .dc.w   1
```

.even 擬似命令が指定されると、偶数バンダリ調整が行われ、次のデータまたは命令は偶数番地に割りあてられます。

1.4.6 条件つきアセンブリ

特定の条件が成立したときのみ、この擬似命令の後続のソース文をアセンブルすることをアセンブラに指示します。
この擬似命令には以下のものがあります。

.if（.ifne）	条件が真のときアセンブル実行
.iff（.ifeq）	条件が偽のときアセンブル実行
.ifdef	シンボルが定義されているときアセンブル実行
.ifndef	シンボルが定義されていないときアセンブル実行
.else	反対の条件が成立するときアセンブル実行
.elseif	反対の条件が成立しかつ特定の条件を満たすときアセンブル実行
.endif（.endc）	条件つきアセンブルの終了

.if 条件つきアセンブル

書式

.if (.ifne)　<式>
　<式>が真のときアセンブル実行

.iff (.ifeq)　<式>
　<式>が偽のときアセンブル実行

.ifdef　<シンボル>
　シンボルが定義されているときアセンブル実行

.ifndef　<シンボル>
　シンボルが定義されていないときアセンブル実行

.else
　反対の条件が存在するときアセンブル実行

.elseif　<式>
　反対の条件が存在し、かつ特定の条件を満たすときアセンブル実行

機能

条件が成り立つときアセンブルを実行します。

解説

いずれも条件つきアセンブル開始文であり、条件の真偽により .endif (.endc) 擬似命令までのソース文をアセンブルするか否かを決定します。
条件つきアセンブリ擬似命令は、アセンブラの開始時に入出力セクションを挿入したり、削除したりする場合によく用いられます。

例

```
.ifdef  symdef
        lea     symdef,a0
        addi    #4,a0
.else
        move    #0,a0
.endif
        jmp     (a0)
```

.endif(.endc)　条件つきアセンブルの終了

書　　式　.endif

.endc

機　　能　条件つきアセンブルを終了させます。

解　　説　条件つきアセンブルの終了を宣言します。

例

```
.ifdef  symdef
        lea      symdef,a0
        addi     #4,a0
.else
        move     #0,a0
.endif
        jmp      (a0)
```

1.4.7 リスティング制御

アセンブルリストの出力に関する制御を行います。
この擬似命令には以下のものがあります。

- .list　アセンブルリストの出力
- .nlist　アセンブルリストの出力の抑止
- .page　アセンブルリストの改頁
- .title　アセンブルリストのタイトルの指定
- .subttl　アセンブルリストのサブタイトルの指定
- .lall　マクロ行の出力
- .sall　マクロ行の出力の抑止

.list アセンブルリストの出力

書　式　.list

機　能　リストファイル作成時に、リスト出力することをアセンブラに指示します。

解　説　リストファイル作成時に、リスト出力することをアセンブラに指示します。
この擬似命令はデフォルトとして設定されています。
.list 擬似命令に続くソース文は、アセンブラが .end または .nlist 擬似命令を検出するまでリスト出力されます。

例

```
start:
*       list
        .list
        move.l  msgptr,a0
        move.l  tblptr,a1
```

.nlist アセンブルリストの出力抑止

書式	.nlist
機能	リストファイル作成時に、リスト出力されないことをアセンブラに指示します。
解説	アセンブラが .list 擬似命令を検出するまで、アセンブルリストの出力を抑制します。

例

```
*       nlist
        .nlist
        move.l  msgptr,a0
        move.l  tblptr,a1
                .
                .
                .
        .list
```

.page　アセンブルリストの改頁

書　式　.page

機　能　アセンブルリストの改頁

解　説　.page擬似命令が検出されたところで改頁し、タイトルやサブタイトルなどを出力します。

例

```
                    .
                    .
                    .
          ori       #Carry,ccr
          rts
          .page

subrtn1:
          movem.l d0-d7/a0-a5,-(sp)
                    .
                    .
                    .
```

.title　アセンブルリストのタイトルの指定

書式　.title　<文字列>

機能　アセンブルリストのタイトルの指定

解説　アセンブルリストの各頁の始めに出力されるタイトルを、<文字列>で指定された文字列に定義します。
<文字列>は1行以内でなければなりません。

例

```
.title This is sample program.
```

上記の擬似命令が記述されているソースプログラムをアセンブルしてリストを出力すると、以下のように各頁の先頭にタイトルを出力します。

```
X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
This is sample program.                    mnt/dd/yy hh:mm:ss
このリストは68000のアセンブリ言語です       Page 1-1
```

.subttl　アセンブルリストのサブタイトルの指定

書式　.subttl　＜文字列＞

機能　アセンブルリストのサブタイトルの指定

解説　アセンブルリストの各頁の始めに出力されるサブタイトルを、＜文字列＞で指定された文字列に定義します。
＜文字列＞は1行以内でなければなりません。

例

```
.subttl このリストは68000のアセンブリ言語です
```

上記の擬似命令が記述されているソースプログラムをアセンブルしてリストを出力すると、以下のように各頁の先頭にサブタイトルを出力します。

```
X68k Assembler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
This is sample program.                    mnt/dd/yy hh:mm:ss
このリストは68000のアセンブリ言語です        Page 1-1
```

.lall マクロ行の出力

書式　.lall

機能　マクロ行の出力

解説　.sall 擬似命令を検出するまで、マクロ行を展開してアセンブルリストに出力します。

マクロ行自体は、.lall, .sall 擬似命令の有無に関係なく出力されます。

.lall 擬似命令がその前に存在していたならば、マクロ行の次にそのマクロの内容を出力します。

例

```
sample  .macro
        ori        #Carry,ccr
        .endm
                .
                .
                .
        .lall
        sample
        move.l    d0,d1
```

上記のソースプログラムをアセンブルしてリストを出力させると、マクロ sample は以下のように出力されます。

```
19 0000000a                           .lall
20 0000000a                           sample
20 0000000a 003c0001                  ori      #Carry,ccr
21 0000000e 2200                      move.l   d0,d1
```

.sall マクロ行の出力の抑止

書　式　.sall

機　能　マクロ行の出力の抑止

解　説　.lall 擬似命令を検出するまで、マクロ行の展開をせずにアセンブルリストに出力しません。

マクロ行自体は、.lall, .sall 擬似命令の有無に関係なく出力されます。

.sall 擬似命令がマクロ行の前に存在していたならば、マクロ行のみを出力します。

例

```
sample  .macro
        ori     #Carry,ccr
        .endm
                .
                .
                .
        .small
        sample
        move.l  d0,d1
```

上記のソースプログラムをアセンブルしてリストを出力させると、マクロ sample は以下のように展開されずに出力されます。

```
19 0000000a                    .sall
20 0000000a                    sample
21 0000000e 2200              move.l  d0,d1
```

1.4.8 シンボリックデバッグ情報の指定

ソースコードデバッグ等でプログラムをデバッグするときに使用されるシンボリックデバッグ情報をオブジェクトに出力するための擬似命令です。
以下に示す擬似命令が、シンボリックデバッグ情報の指定のために用意されています。

- .file　ソースファイル名の出力指定
- .ln　行番号とロケーションの対応の出力指定
- .def　シンボルテーブルエントリの作成開始
- .endef　シンボルテーブルエントリの終了

さらに、.def, .endef 擬似命令により作成されるシンボルテーブルエントリに格納するシンボル情報を指定する擬似命令があります。
これらの擬似命令には、以下のものがあります。

- .val　シンボルの値の指定
- .scl　記憶クラスの宣言
- .type　C 言語における型の宣言
- .tag　タグ名の宣言
- .line　行番号の指定
- .size　サイズの指定
- .dim　配列の指定

.file　ソースファイル名の出力指定

書　　式　.file　"<ファイル名>"

機　　能　ソースファイル名をオブジェクトに出力します。

解　　説　ソースファイル名をオブジェクトに出力します。
<ファイル名>はＣ言語で記述されたソースファイルを指定します。

例

```
.file  "sample.c"
```

.ln　行番号とロケーションの対応の出力指定

書　式

.ln　＜行番号＞［,＜ロケーション値＞］

機　能

行番号とロケーションの対応をオブジェクトに出力します。

解　説

行番号とロケーションの対応をオブジェクトに出力します。
＜行番号＞には、C 言語で記述されたソースプログラムにおける行番号を指定します。
＜ロケーション値＞には、オブジェクトの先頭からのロケーションの値を指定します。省略すると、この擬似命令が存在するところのロケーションが設定されます。

例

```
.ln　10
```

.def, .endef　シンボルテーブルエントリの作成

書　式

.def　<シンボル名>
（属性を指定する擬似命令）
.endef

機　能

シンボルテーブルエントリを作成します。

解　説

C言語における変数や関数の属性などを格納するシンボルテーブルエントリを作成します。
<シンボル名>には、シンボルテーブルエントリを作成したいシンボルの名前を指定します。
シンボルテーブルはこの.def擬似命令の間にある、シンボルの属性を指定する擬似命令をもとに作成します。

例

```
.def     _stach
.vzl     _stach
.scl     2
.type    50
.endef
```

.val シンボルの値の指定

書　式　.val　＜式＞

機　能　式の値をシンボルのものとします。

.scl 記憶クラスの宣言

書　式　.scl　＜式＞

機　能　記憶域クラスを宣言します。

.type C言語における型の宣言

書　式　.type　＜式＞

機　能　C言語における型を宣言します。

.tag タグ名の宣言

書　式　.tag　＜タグ名＞

機　能　タグ名を宣言します。

.line　行番号の指定

書　式　.line　<式>

機　能　シンボルに行番号を与えます。

.size　サイズの指定

書　式　.size　<式>

機　能　シンボルにサイズを与えます。

.dim　配列の指定

書　式　.dim　<式1> [, <式2>・・・・]

機　能　シンボルに配列名とその次元数を与えます。

第2章

命令セットリファレンス一覧

命令セットリファレンス一覧表の見方
命令セットリファレンス一覧

2.1 命令セットリファレンス一覧表の見方

この章では、MPU68000の命令セットを、データ移動命令、整数計算命令、論理演算命令、シフト・ローテート命令、ビット操作命令、BCD演算命令、プログラム制御命令、システム制御命令に分類して掲載してあります。
実際にアセンブリ言語によってプログラムを記述する方法は、本書の「第1部　ユーザーズガイド」を参照してください。

●「命令」について

オペレーションが似ている命令ごとに分けて、表にしてあります。
1つの表の中では、各命令のニーモニックについて上からアルファベット順に列挙しています。

●「サイズ」について

命令が扱うオペランドのデータサイズを指定する場合に使用します。

B	バイト・サイズ（8ビット）
W	ワード・サイズ（16ビット）
L	ロングワード・サイズ（32ビット）

●「オペランド」について

命令が扱うデータがレジスタであるのか、メモリのどのアドレスにあるのかを指定します。
なお値を読み出すものがソース、結果を置くものがディスティネーションです。

Dm, Dn	データレジスタ（D0～D7）
Am, An	アドレスレジスタ（A0～A7）
＜sea＞	ソースの実効アドレス
＜dea＞	デスティネーションの実効アドレス
＜ea＞	実効アドレス
＜regs＞	レジスタリスト（例：D0-D5／A0／A4）
＜data＞	イミディエートデータ、定数
＜label＞	ラベル（通常は分岐先やジャンプ先を示す）
＜bn＞	ビット位置を指し示すオペランド
＜vec＞	ベクタ番号（ここでは0～15）

2.1 命令セットリファレンス一覧表の見方

●「アドレッシング・モード」について

MPU68000で扱うことができるアドレッシングモードは次の通りです。

Dn	データレジスタ直接モード
An	アドレスレジスタ直接モード
(An)	アドレスレジスタ間接モード
(An)+	ポストインクリメント付きアドレスレジスタ間接モード
−(An)	プリデクリメント付きアドレスレジスタ間接モード
d16 (An)	ディスプレースメント付きアドレスレジスタ間接モード ディスプレースメントのサイズは16ビット
d8 (An, Xn)	ディスプレースメント&インデックス付きアドレスレジスタ間接モード ディスプレースメントのサイズは8ビット インデックスはデータまたはアドレスレジスタ そのサイズはワードかロングワード
ABS	絶対アドレスモード ショートアドレスかロングアドレス
#IMM	イミディエイトモード
d16 (PC)	ディスプレースメント付きPC相対間接モード ディスプレースメントのサイズは16ビット
d8 (PC, Xn)	ディスプレースメント&インデックス付きPC相対間接モード ディスプレースメントのサイズは8ビット インデックスはデータまたはアドレスレジスタ そのサイズはワードかロングワード

●「コンディション・コード」について

命令を実行した際に、影響を受けるコンディション・コードを表しています。表中の略号と状態は次の通りです。

C	キャリーを示すビット
V	オーバーフローを表すビット
Z	ゼロであることを表すビット
N	負の値であることを表すビット
X	拡張ビット
*	影響あり（前の状態から変化する）
—	影響なし（前の状態から変化しない）
0	クリア

2.1　命令セットリファレンス一覧表の見方

1	セット
U	定義されていない（不定）

● **「解説」について**

各命令の簡単な動作説明を記してあります。

2.2 命令セットリファレンス一覧

●データ移動命令

命令	サイズ			オペランド		アドレッシングモード						
	B	W	L			Dn	An	(An)	(An)+	−(An)	d16 (An)	d8 (An, Xn)
EXG			L	<sea>, <dea>	<sea>	○	○					
					<dea>	○	○					
LEA			L	<sea>, An	<sea>			○			○	○
LINK				An, #<data>	<data>							
MOVE	B	W	L	<sea>, <dea>	<sea>	○	○	○	○	○	○	○
					<dea>	○		○	○	○	○	○
MOVEA		W	L	<sea>, An	<sea>	○	○	○	○	○	○	○
MOVEM		W	L	<sea>, <regs>	<sea>			○	○		○	○
					<regs>	○	○					
		W	L	<regs>, <dea>	<dea>			○		○	○	○
					<regs>	○	○					
MOVEP		W	L	<sea>, An	<sea>						○	
		W	L	Dn, <dea>	<dea>						○	
MOVEQ			L	#<data>, Dn	<data>							
PEA			L	<sea>	<sea>			○			○	○
UNLK				An								

●整数計算命令

命令	サイズ			オペランド		アドレッシングモード						
	B	W	L			Dn	An	(An)	(An)+	−(An)	d16 (An)	d8 (An, Xn)
ADD	B	W	L	<sea>, Dn	<sea>	○	LW	○	○	○	○	○
	B	W	L	Dn, <dea>	<dea>			○	○	○	○	○
ADDA		W	L	<sea>, An	<sea>	○	○	○	○	○	○	○
ADDI	B	W	L	#<data>, <dea>	<dea>	○		○	○	○	○	○
					<data>							
ADDQ	B	W	L	#<data>, <dea>	<dea>	○	LW	○	○	○	○	○
					<data>							
ADDX	B	W	L	Dm, Dn								
	B	W	L	−(Am), −(An)								
CLR	B	W	L	<dea>	<dea>	○		○	○	○	○	○
CMP	B	W	L	<sea>, Dn	<sea>	○	LW	○	○	○	○	○
CMPA		W	L	<sea>, An	<sea>	○	○	○	○	○	○	○
CMPI	B	W	L	#<data>, <dea>	<dea>	○		○	○	○	○	○
					<data>							
CMPM	B	W	L	(Am)+, (An)+								
DIVS		W		<sea>, Dn	<sea>	○		○	○	○	○	○
DIVU		W		<sea>, Dn	<sea>	○		○	○	○	○	○
EXT		W	L	Dn								
MULS		W		<sea>, Dn	<sea>	○		○	○	○	○	○
MULU		W		<sea>, Dn	<sea>	○		○	○	○	○	○
NEG	B	W	L	<dea>	<dea>	○		○	○	○	○	○
NEGX	B	W	L	<dea>	<dea>	○		○	○	○	○	○
SUB	B	W	L	<sea>, Dn	<sea>	○	LW	○	○	○	○	○
	B	W	L	Dn, <dea>	<dea>			○	○	○	○	○
SUBA		W	L	<sea>, An	<sea>	○	○	○	○	○	○	○
SUBI	B	W	L	#<data>, <dea>	<dea>	○		○	○	○	○	○
					<data>							
SUBQ	B	W	L	#<data>, <dea>	<dea>	○	LW	○	○	○	○	○
					<data>							
SUBX	B	W	L	Dm, Dn								
	B	W	L	−(Am), −(An)								

注）アドレッシングモードのAnにおける表記"LW"は、「ワード又はロングワードのサイズのみ許される」という意味です。

2.2　命令セットリファレンス一覧

データ移動命令

				コンディション・コード					解　説
ABS	#IMM	d16 (PC)	d8 (PC, Xn)	X	N	Z	V	C	
				—	—	—	—	—	レジスタ交換
○		○	○	—	—	—	—	—	実効アドレスのロード
	−32768～0			—	—	—	—	—	リンクと割付け
○	○	○	○	—	*	*	0	0	データ移動
○									
○	○	○	○	—	—	—	—	—	アドレスデータの移動
○		○	○	—	—	—	—	—	複数レジスタのデータ移動
	○								
○				—	—	—	—	—	
	○								
				—	—	—	—	—	周辺I/Oとのデータ転送
				—	—	—	—	—	
	−128～127			—	*	*	0	0	高速なデータ移動
○		○	○	—	—	—	—	—	実効アドレスのスタックへの移動
				—	—	—	—	—	リンク解除

整数計算命令

				コンディション・コード					解　説
ABS	#IMM	d16 (PC)	d8 (PC, Xn)	X	N	Z	V	C	
○	○	○	○	*	*	*	*	*	加算
○				*	*	*	*	*	
○	○	○	○	—	—	—	—	—	アドレスデータの加算
○				*	*	*	*	*	指定数値との加算
	○								
○				*	*	*	*	*	指定数値との高速加算
	1～8								
				*	*	*	*	*	拡張加算
				*	*	*	*	*	
○				—	0	1	0	0	オペランドの0クリア
○	○	○	○	—	*	*	*	*	比較
○	○	○	○	—	*	*	*	*	アドレスデータの比較
○				—	*	*	*	*	指定数値との比較
	○								
				—	*	*	*	*	メモリ内容の比較
○	○	○	○	—	*	*	*	0	符号付き除算
○	○	○	○	—	*	*	*	0	符号なし除算
				—	*	*	0	0	符号拡張
○	○	○	○	—	*	*	0	0	符号付き乗算
○	○	○	○	—	*	*	0	0	符号なし乗算
○				*	*	*	*	*	符号反転
○				*	*	*	*	*	拡張付き符号反転
○	○	○	○	*	*	*	*	*	減算
○				*	*	*	*	*	
○	○	○	○	—	—	—	—	—	アドレスデータとの減算
○				*	*	*	*	*	指定数値との減算
○									
○				*	*	*	*	*	指定数値との高速減算
	1～8								
				*	*	*	*	*	拡張減算
				*	*	*	*	*	

2.2 命令セットリファレンス一覧

●論理演算命令

命令	サイズ			オペランド		アドレッシングモード						
	B	W	L			Dn	An	(An)	(An)+	−(An)	d16 (An)	d8 (An, Xn)
AND	B	W	L	<sea>, Dn	<sea>	○		○	○	○	○	○
	B	W	L	Dn, <dea>	<dea>			○	○	○	○	○
ANDI	B	W	L	#<data>, <dea>	<dea>	○		○	○	○	○	○
					<data>							
EOR	B	W	L	Dn, <dea>	<dea>	○		○	○	○	○	○
EORI	B	W	L	#<data>, <dea>	<dea>	○		○	○	○	○	○
					<data>							
NOT	B	W	L	<dea>	<dea>	○		○	○	○	○	○
OR	B	W	L	<sea>, Dn	<sea>	○		○	○	○	○	○
	B	W	L	Dn, <dea>	<dea>			○	○	○	○	
ORI	B	W	L	#<data>, <dea>	<dea>	○		○	○	○	○	○
					<data>							
TST	B	W	L	<dea>	<dea>	○		○	○	○	○	○

●シフト・ローテート操作命令

命令	サイズ			オペランド		アドレッシングモード						
	B	W	L			Dn	An	(An)	(An)+	−(An)	d16 (An)	d8 (An, Xn)
ASL	B	W	L	Dm, Dn								
	B	W	L	#<data>, Dn	<data>							
		W		<dea>	<dea>			○	○	○	○	○
ASR	B	W	L	Dm, Dn								
	B	W	L	#<data>, Dn	<data>							
		W		<dea>	<dea>			○	○	○	○	○
LSL	B	W	L	Dm, Dn								
	B	W	L	#<data>, Dn	<data>							
		W		<dea>	<dea>			○	○	○	○	○
LSR	B	W	L	Dm, Dn								
	B	W	L	#<data>, Dn	<data>							
		W		<dea>	<dea>			○	○	○	○	○
ROL	B	W	L	Dm, Dn								
	B	W	L	#<data>, Dn	<data>							
		W		<dea>	<dea>			○	○	○	○	○
ROR	B	W	L	Dm, Dn								
	B	W	L	#<data>, Dn	<data>							
		W		<dea>	<dea>			○	○	○	○	○
ROXL	B	W	L	Dm, Dn								
	B	W	L	#<data>, Dn	<data>							
		W		<dea>	<dea>			○	○	○	○	○
ROXR	B	W	L	Dm, Dn								
	B	W	L	#<data>, Dn	<data>							
		W		<dea>	<dea>			○	○	○	○	○
SWAP		W		Dn								

2.2　命令セットリファレンス一覧

論理演算命令

				コンディション・コード					解　説
ABS	#IMM	d16 (PC)	d8 (PC, Xn)	X	N	Z	V	C	
○	○	○	○	—	*	*	0	0	論理積
○				—	*	*	0	0	
○				—	*	*	0	0	指定数値との論理積
	○								
○				—	*	*	0	0	排他的論理和
○				—	*	*	0	0	指定数値との排他的論理和
	○								
○				—	*	*	0	0	論理反転
○	○	○	○	—	*	*	0	0	論理和
○				—	*	*	0	0	
○				—	*	*	0	0	指定数値との論理和
	○								
○				—	*	*	0	0	オペランドのテスト

シフト・ローテート命令

				コンディション・コード					解　説
ABS	#IMM	d16 (PC)	d8 (PC, Xn)	X	N	Z	V	C	
				*	*	*	*	*	左への算術シフト
	1～8			*	*	*	*	*	
○				*	*	*	*	*	
				*	*	*	*	*	右への算術シフト
	1～8			*	*	*	*	*	
○				*	*	*	*	*	
				*	*	*	0	*	左への論理シフト
	1～8			*	*	*	0	*	
○				*	*	*	0	*	
				*	*	*	0	*	右への論理シフト
	1～8			*	*	*	0	*	
○				*	*	*	0	*	
				—	*	*	0	*	左へのローテート
	1～8			—	*	*	0	*	
○				—	*	*	0	*	
				—	*	*	0	*	右へのローテート
	1～8			—	*	*	0	*	
○				—	*	*	0	*	
				*	*	*	0	*	拡張付きの左へのローテート
	1～8			*	*	*	0	*	
○				*	*	*	0	*	
				*	*	*	0	*	拡張付きの右へのローテート
	1～8			*	*	*	0	*	
○				*	*	*	0	*	
				—	*	*	0	0	レジスタの上位／下位ワードの交換

2.2 命令セットリファレンス一覧

●ビット操作命令

命令	サイズ			オペランド		アドレッシングモード						
	B	W	L			Dn	An	(An)	(An)+	−(An)	d16 (An)	d8 (An, Xn)
BCHG			L	<bn>, Dn	<bn>	○						
	B			<bn>, <dea>	<bn>	○						
					<dea>			○	○	○	○	○
BCLR			L	<bn>, Dn	<bn>	○						
	B			<bn>, <dea>	<bn>	○						
					<dea>			○	○	○	○	○
BSET			L	<bn>, Dn	<bn>	○						
	B			<bn>, <dea>	<bn>	○						
					<dea>			○	○	○	○	○
BTST			L	<bn>, Dn	<bn>	○						
	B			<bn>, <dea>	<bn>	○						
					<dea>			○	○	○	○	○

● BCD 演算命令

命令	サイズ			オペランド		アドレッシングモード						
	B	W	L			Dn	An	(An)	(An)+	−(An)	d16 (An)	d8 (An, Xn)
ABCD	B			Dm, Dn								
	B			−(Am), −(An)								
NBCD	B			<dea>	<dea>	○		○	○	○	○	○
SBCD	B			Dm, Dn								
	B			−(Am), −(An)								

●プログラム制御命令

命令	サイズ			オペランド		アドレッシングモード						
	B	W	L			Dn	An	(An)	(An)+	−(An)	d16 (An)	d8 (An, Xn)
Bcc	B	W		<label>								
BRA	B	W		<label>								
BSR	B	W		<label>								
DBcc		W		Dn, <label>								
JMP				<ea>	<ea>			○			○	○
JSR				<ea>	<ea>			○			○	○
NOP												
RTR												
RTS												
Scc	B			<ea>	<ea>	○		○	○	○	○	○

cc に従った分岐

条件	cc	
	符号あり	符号なし
より大きい	HI	GT
より大きいか等しい	CC	GE
より小さいか等しい	LS	LE
より小さい	CS	LT
等しい	EQ	
等しくない	NE	
正の値	PL	--
負の数	MI	--
オーバーフローである	VS	
オーバーフローでない	VC	

2.2 命令セットリファレンス一覧

ビット操作命令

				コンディション・コード					解説
ABS	#IMM	d16 (PC)	d8 (PC, Xn)	X	N	Z	V	C	
	0~31			—	—	*	—	—	ビットのテストと反転
	0~8			—	—	*	—	—	
○									
	0~31			—	—	*	—	—	ビットのテストと0クリア
	0~8			—	—	*	—	—	
○									
	0~31			—	—	*	—	—	ビットのテストと1セット
	0~8			—	—	*	—	—	
○									
	0~31			—	—	*	—	—	ビットのテスト
	0~8			—	—	*	—	—	
○									

BCD 演算命令

				コンディション・コード					解説
ABS	#IMM	d16 (PC)	d8 (PC, Xn)	X	N	Z	V	C	
				*	U	*	U	*	BCD 加算
				*	U	*	U	*	
○				*	U	*	U	*	BCD 符号反転
				*	U	*	U	*	BCD 減算
				*	U	*	U	*	

プログラム制御命令

				コンディション・コード					解説
ABS	#IMM	d16 (PC)	d8 (PC, Xn)	X	N	Z	V	C	
				—	—	—	—	—	cc に従った分岐
				—	—	—	—	—	無条件分岐
				—	—	—	—	—	サブルーチンへの分岐
				—	—	—	—	—	ループ命令
○		○	○	—	—	—	—	—	ジャンプ
○		○	○	—	—	—	—	—	サブルーチンへのジャンプ
				—	—	—	—	—	ノー・オペレーション
				*	*	*	*	*	CCR の回復とリターン
				—	—	—	—	—	サブルーチンからのリターン
○				—	—	—	—	—	cc に従ったセット／リセット

2.2　命令セットリファレンス一覧

●システム制御命令

命令	サイズ			オペランド		アドレッシングモード						
	B	W	L			Dn	An	(An)	(An)+	−(An)	d16 (An)	d8 (An, Xn)
ANDI		W		#<data>, sr	<data>							
ANDI	B			#<data>, ccr	<data>							
EORI		W		#<data>, sr	<data>							
EORI	B			#<data>, ccr	<data>							
MOVE	B			<sea>, ccr	<sea	○		○	○	○	○	○
	B			ccr, <dea>	<dea>	○		○	○	○	○	○
		W		<sea>, sr	<sea>	○		○	○	○	○	○
		W		sr, <dea>	<dea>	○		○	○	○	○	○
			L	An, USP								
ORI		W		#<data>, sr	<data>							
ORI	B			#<data>, ccr	<data>							
RESET												
RTE												
CHK		W		<sea>, Dn	<sea>	○		○	○	○	○	○
STOP				#<data>	<data>							
TAS	B			<ea>	<ea>	○		○	○	○	○	○
TRAP				#<vec>	<vec>							
TRAPV												

2.2　命令セットリファレンス一覧

システム制御命令

				コンディション・コード					解　説
ABS	#IMM	d16 (PC)	d8 (PC, Xn)	X	N	Z	V	C	
	○			*	*	*	*	*	指定数値とSRとの論理積（特権命令）
	○			*	*	*	*	*	指定数値とCCRとの論理積
	○			*	*	*	*	*	指定数値とSRとの排他的論理和（特権命令）
	○			*	*	*	*	*	指定数値とCCRとの排他的論理和
○	○	○	○	*	*	*	*	*	CCRへのデータ移動
○				—	—	—	—	—	CCRからのデータ移動
○	○	○	○	*	*	*	*	*	SRへのデータ移動（特権命令）
○				—	—	—	—	—	SRからのデータ移動
○				—	—	—	—	—	USPへのデータ移動（特権命令）
	○			*	*	*	*	*	指定数値とSRとの論理和（特権命令）
	○			*	*	*	*	*	指定数値とCCRとの論理和
				—	—	—	—	—	外部デバイスのリセット（特権命令）
				*	*	*	*	*	例外処理からのリターン（特権命令）
○	○	○	○	—	*	U	U	U	レジスタの境界チェック
	0～65535			*	*	*	*	*	SRのロードとストップ（特権命令）
	○			—	*	*	0	0	オペランドのテストとセット
	0～15			—	—	—	—	—	トラップ
				—	—	—	—	—	オーバーフローによるトラップ

索引 50音順

ア

カ

マ

ラ

アルファベット順

シャープ株式会社
本　　　社　〒545 大阪市阿倍野区長池町22番22号
電子機器事業本部　〒329-21 栃木県矢板市早川町174番地
液晶映像システム事業部　第2商品企画部
お問い合わせ先　〒162 東京都新宿区市谷八幡町8番地　電話 (03)3260-1161(大代表)
東京支社内　液晶映像システム事業部　第2商品企画部　ソフトウェア担当